Méthodologie et environnement de développement orientés objets : de l'analyse mathématique à la programmation

S. Labbé, J. Laminie, V. Louvet
Laboratoire de Mathématique, Analyse Numérique et EDP.
Université Paris Sud, Orsay

6 janvier 2009
Version 1.1

Table des matières

Avertissement **ix**

Introduction **1**

I De l'analyse mathématique à la conception **3**

1 Philosophie et motivations **5**
1.1 Environnement 5
1.1.1 La recherche 6
1.1.2 L'enseignement 6
1.1.3 L'environnement informatique 7
1.1.4 En conclusion 8
1.2 Objetifs 8
1.3 Etat de l'art 9
1.4 Méthodologie 10

2 Analyse mathématique **11**
2.1 Cadre théorique 12
2.2 Analyse 13
2.3 Conception préliminaire 16
2.4 Conception détaillée 16
2.4.1 Conception détaillée du paquetage `Problème` 18
2.4.2 Conception détaillée du paquetage `Opérateur` 19
2.4.3 Conception détaillée du paquetage `Variable` 22
2.4.4 Conception détaillée du paquetage `Domaine` 22
2.4.5 Conception détaillée de la hiérarchie `Discrétisation` 23
2.4.6 Conception détaillée de la hiérarchie `Stockage` 24
2.4.7 Conception détaillée des classes utilitaires 24

3 Environnement **27**
3.1 Expression du besoin 27
3.2 Etude préliminaire 28
3.2.1 Environnement de travail 28
3.2.2 Acteurs 28
3.2.3 Interactions 29
3.2.4 Diagramme de contexte 29
3.3 Conceptualisation 29
3.3.1 UC niveau tâche 30
3.3.2 UC niveau fonction 30
3.3.3 Diagramme de cas d'utilisation 31
3.3.4 Spécification du besoin relatif à l'interface graphique 33

3.4 Analyse . . . 34
3.4.1 Analyse du domaine . . . 34
3.4.2 Analyse applicative . . . 35
3.4.3 Modèle d'analyse . . . 39
3.5 Architecture . . . 40
3.5.1 Architecture logicielle . . . 40
3.5.2 Architecture matérielle . . . 41
3.6 Conception préliminaire . . . 42
3.6.1 Module User . . . 43
3.6.2 Module Data . . . 43
3.6.3 Module Execution . . . 43
3.6.4 Module View . . . 44
3.6.5 Module Code . . . 48
3.7 Conception détaillée . . . 48
3.7.1 Conception détaillée du sous-système serveur de données . . . 50
3.7.2 Conception détaillée du sous-système code de calcul . . . 54
3.7.3 Conception détaillée du sous-système utilisateur final . . . 54
3.8 Implémentation et tests . . . 55
3.8.1 Choix techniques pour l'implémentation . . . 55
3.8.2 Modèle d'implémentation : diagramme de composants . . . 56
3.8.3 Tests . . . 59
3.9 Itération du processus . . . 59
3.9.1 Scénarii secondaires des cas d'utilisation . . . 60
3.9.2 Nouvelles fonctionnalités . . . 60

II De la conception à la programmation 61

4 Application 63
4.1 Discrétisation par éléments finis . . . 63
4.2 Discrétisation par différences finis . . . 65
4.3 Mise en œuvre . . . 67
4.3.1 Définition du problème . . . 67
4.3.2 Le paquetage `Problem` . . . 67
4.3.3 Le paquetage `Variable` . . . 69
4.3.4 Le paquetage `Operateur` . . . 69
4.3.5 Le paquetage `Domain` . . . 69
4.3.6 La classe `Method` . . . 71
4.3.7 Eléments finis . . . 71
4.3.8 Différences finis . . . 74
4.3.9 La classe `Storage` . . . 75
4.3.10 Construction informatique du problème elliptique . . . 76

5 Environnement de travail 79
5.1 Démarrage du serveur . . . 79
5.1.1 Services CORBA . . . 79
5.1.2 Serveur . . . 80
5.2 Instrumentation . . . 80
5.2.1 Langages disponibles . . . 80
5.2.2 Compilation . . . 80
5.2.3 Initialisations, finalisation . . . 81
5.2.4 Sortie standart : Cout . . . 82
5.2.5 Profiling du code : Flow trace . . . 85
5.2.6 Instrumentation d'un fichier source . . . 86

5.3 Exploitation graphique . . . 89
5.3.1 Démarrage de l'interface . . . 89
5.3.2 Fenêtre principale d'accueil . . . 89
5.3.3 Fenêtre de connexion . . . 89
5.3.4 Fenêtre des données utilisateur . . . 90
5.3.5 Fenêtre des calculs . . . 90
5.3.6 Fenêtre d'une exécution particulière . . . 91
5.3.7 Fenêtre d'un flux particulier . . . 91
5.3.8 Fenêtre de documentation . . . 94

6 Documentation **97**
6.1 Niveau de documentation . . . 97
6.2 Structuration de la documentation . . . 98
6.2.1 Structure de l'entête . . . 98
6.2.2 Structure du corps du programme . . . 98
6.3 Utilisation du programme . . . 99
6.3.1 Mode en ligne . . . 99
6.3.2 Mode graphique . . . 99
6.4 Exemples . . . 99
6.4.1 Exemple en fortran 90 . . . 100
6.4.2 Exemple en C++ . . . 101

Bibliographie **107**

Annexes **107**

A Etat de l'art **107**
A.1 Les critères . . . 107
A.2 Logiciels et Bibliothèques Objets . . . 107
A.2.1 elsA . . . 108
A.2.2 FreeFEM++ . . . 108
A.2.3 Mélina . . . 108
A.2.4 Overture . . . 108
A.3 Logiciels et Bibliothèques NON Objets . . . 108
A.3.1 Les bibliothéques d'algèbre linéaire . . . 109
A.3.2 Les autres bibliothèques . . . 110
A.4 Les classes objets pour la gestion des tableaux . . . 110
A.4.1 Le standard de C++ . . . 110
A.4.2 Le standard de JAVA . . . 110
A.4.3 TNT . . . 111
A.4.4 A++/P++ . . . 111
A.5 Les mailleurs . . . 111
A.6 Les bibliothèques d'outils parallèles . . . 111
A.6.1 Bibliothèques de synchronisation . . . 111
A.6.2 Les bibliothèques de passages de messages . . . 112

B Notations UML **113**
B.1 Paquetages . . . 114
B.2 Classes et Visibilité . . . 114
B.3 Associations et cardinalité . . . 115
B.4 Généralisation et héritage . . . 116
B.5 Conclusions . . . 116

C CORBA : mécanismes de fonctionnement **117**
C.1 Concepts et notions 117
C.2 Fonctionnement du bus d'objets répartis et du processus d'invocation des requêtes 121
C.3 Utilisation des services CORBA 122
C.3.1 Le service de Nommage (Naming Service) 122
C.3.2 Le service des Evènements (Event Service) 123

Table des figures

2.2 Diagramme de séquence global représentant la résolution d'un problème mathématique. . . . 14
2.1 Diagramme de collaboration global de la résolution. . . . 15
2.4 Diagramme de séquence global représentant la résolution d'un problème mathématique au niveau des opérateurs. . . . 16
2.3 Modèle d'analyse du problème mathématique. . . . 17
2.5 Diagramme de classes du paquetage `Problème` . . . 20
2.6 Diagramme de classes du paquetage `Opérateur`. . . . 21
2.7 Diagramme de classes du paquetage `Variable`. . . . 22
2.8 Diagramme de classes du paquetage `Domaine`. . . . 23
2.9 Diagramme de classes de la hiérarchie `Discrétisation`. . . . 24
2.10 Diagramme de classes de la hiérarchie `Stockage`. . . . 24
2.11 Diagramme de composants illustrant les communications entres les classes mathématiques et les classes utilitaires. . . . 25

3.1 Environnement de travail global. . . . 28
3.2 Diagramme de contexte. . . . 29
3.3 Diagramme des cas d'utilisation. . . . 32
3.4 Diagramme U.M.L. du besoin IHM. . . . 33
3.6 Diagramme de séquence du cas d'utilisation 4. . . . 35
3.5 Diagramme des classes issues de l'analyse du domaine. . . . 36
3.7 Diagramme de séquence du cas d'utilisation 5. . . . 37
3.8 Diagramme de séquence du cas d'utilisation 6. . . . 37
3.9 Diagramme de séquence du cas d'utilisation 2. . . . 38
3.10 Diagramme de séquence du cas d'utilisation 3. . . . 38
3.11 Diagramme de classes du modèle d'analyse. . . . 39
3.12 Modèle d'analyse projeté sur une architecture de type 3 tiers. . . . 40
3.13 Schéma de l'architecture physique. . . . 42
3.14 Diagramme de déploiement sur l'architecture. . . . 43
3.15 Diagramme U.M.L. du sous-système *User*. . . . 44
3.16 Diagramme U.M.L. du sous-système *Data*. . . . 45
3.17 Diagramme U.M.L. du sous-système *Execution*. . . . 46
3.18 Diagramme U.M.L. des classes générales de vue et de gestion. . . . 46
3.19 Diagramme U.M.L. des classes graphiques liées à l'objet utilisateur. . . . 47
3.20 Diagramme U.M.L. des classes graphiques liées aux pages jaunes des exécutions. . . . 47
3.21 Diagramme U.M.L. des classes graphiques liées à une exécution. . . . 48
3.22 Diagramme U.M.L. des classes graphiques liées aux données. . . . 49
3.23 Diagramme U.M.L. des classes issues de la conception de la partie instrumentation du code de calcul. . . . 49
3.24 Diagramme des classes Corba du serveur. . . . 51
3.25 Diagramme des interfaces IDL des objets communiquants. . . . 52
3.26 Diagramme des interfaces IDL des fabriques d'objets communiquants. . . . 52
3.27 Gestion des contextes de nom relatifs aux données et exécutions. . . . 53

3.28 Gestion du contexte de nom racine. 53
3.29 Classes Corba du client. 54
3.30 Diagramme des classes de gestion d'évènements. 54
3.31 Diagramme de composants des entités relatives aux objets distribués. 56
3.32 Diagramme de composants relatif à la base de données utilisateurs. 57
3.33 Diagramme de composants relatif aux programmes clients. 57
3.34 Diagramme de composants relatif aux librairies extérieures. 58
3.35 Diagramme de composants relatif à l'exécutable du serveur. 58
3.36 Diagramme de composants relatif aux exécutables des codes de calcul. 58
3.37 Diagramme de composants relatif à l'exécution de l'interface graphiquel. 59

4.1 Classe Problem. 68
4.2 Classe Discret_Problem. 68
4.3 Classe Algebric_Problem. 69
4.4 Classe DiscreteDomain. 70
4.5 Hiérarchie de la classe `Method` relative aux éléments finis. 73
4.6 Hiérarchie de la classe `Method` relative aux différences finies. 74
4.7 Hiérarchie de la classe `Storage`. 75

5.1 Fenêtre d'accueil et de connexion de l'interface graphique. 90
5.2 Fenêtre donnant les informations de l'utilisateur connecté et des calculs accessibles par l'interface graphique. 90
5.3 Fenêtre donnant les informations d'un calcul particulier accessible par l'interface graphique. 91
5.4 Fenêtre donnant les informations de flux issus d'un calcul accessible par l'interface graphique. 92
5.5 Fenêtre permettant la visualisation textuelle d'un flux par l'interface graphique. . . 92
5.6 Fenêtre permettant le choix des données pour la visualisation graphique d'un flux via l'interface graphique. 93
5.7 Fenêtre permettant la visualisation graphique (courbe 2D) d'un flux via l'interface graphique. 94
5.8 Fenêtre permettant la visualisation de la documentation d'un code via l'interface graphique. 94

6.1 Fenêtre permettant le choix du fichier pour la génération de la documentation . . . 100
6.2 Visualisation de la documentation générée pour le code source en fortran 90 : 1ère partie 102
6.3 Visualisation de la documentation générée pour le code source en fortran 90 : 2ème partie 103
6.4 Visualisation de la documentation générée pour le code source en C++ : 1ère partie 105
6.5 Visualisation de la documentation générée pour le code source en C++ : 2ème partie 106

B.1 Modélisation UML d'un paquetage. 114
B.2 Modélisation UML d'une classe. 115
B.3 Modélisation UML d'une association entre classes. 115
B.4 Modélisation UML d'une agrégation et d'une composition. 116
B.5 Modélisation UML d'une relation de généralisation. 116

C.1 Constitution de l'OMA. 118
C.2 Processus d'invocation des requêtes. 121

Avertissement

Ce document est une première version du projet MOOCS (Méthodologie Orientée Objet pour le Calcul Scientifique). Il est naturellement destiné à évoluer. Des mises à jour seront régulièrement disponibles sur le site http ://www.math.u-psud.fr/~gtoocs.

Introduction

Ce projet a été initié au sein d'un groupe de travail du Laboratoire de Mathématique de l'Université Paris Sud d'Orsay (Groupe de Travail de programmation Orienté Objet pour le Calcul Scientifique, GTOOCS) créé en 1999/2000. Les réflexions menées portaient avant tout sur l'étude de la conception objet appliquée au domaine particulier du calcul scientifique.

A cette occasion, nous remercions pour leur collaboration Frédéric Pascal [1] (qui avec l'un des auteur a écrit une première réflexion sur la programmation objet appliquée au calcul scientifique pour les éléments finis en Fortran 90), Marc Tajchman[2] et Maxime Pallud [3], ainsi qu'Alain Lichnewsky [1] pour ses conseils.

Le but de ce travail est la réalisation d'un cadre de programmation objet pour le calcul scientifique principalement appliqué à la résolution d'équations ou de systèmes d'équations aux dérivées partielles. Les motivations qui nous ont conduits à concevoir ce projet sont nombreuses. Tout d'abord, la complexité des infrastructures informatiques et celle des besoins des utilisateurs et des concepteurs de logiciels fait qu'il est impératif que la technologie de conception permettent la prise en compte efficace des notions comme le développement coopératif ou l'hétérogénéité des matériels. De plus, en tant que laboratoire universitaire, nous devons prendre en considération les besoins d'outils pédagogiques pour la formation des étudiants ou encore ceux de pérénisation des codes de recherches.

La réalisation de codes de calcul scientifique dans le cadre d'un travail de recherche en analyse numérique a pour but l'étude et l'optimisation des méthodes utilisées. Cette approche diffère de la situation classique pour laquelle la finalité est l'obtention d'un résultat. La recherche dans le domaine du calcul scientifique se situe à de nombreux niveaux. Outre la résolution du problème, les numériciens travaillent sur la mathématique des équations, la méthodologie de résolution, les discrétisations mais aussi les sciences de plus bas niveau comme l'algèbre linéaire.

Par ailleurs et d'un point de vue purement informatique, nous sommes conduit à travailler dans un environnement multi-plateformes et multi-langages. Enfin, l'analyse des résultats des codes, au niveau souhaité par les numériciens, implique la mise en œuvre d'outils de développements, d'exploitation et de dépouillement.

Dans ce projet nous nous sommes fixés deux objectifs principaux : avoir une méthodologie de programmation orientée objet pour les mathématiques, ainsi que pour la gestion de l'environnement et la visualisation des résultats. Pour cela, nous présentons dans ce rapport un formalisme mathématique compatible avec la programmation orientée objet ; nous détaillons les liens entre les objets mathématiques issus de ce formalisme et les classes qui les modélisent. Précisons que le but du projet n'est pas la création d'un générateur de code pour la résolution d'équations mais la mise à disposition d'un standart de programmation qui permet de faciliter le développement de codes de calcul répondant aux critéres cités plus haut. Cette méthodologie a déjà été appliquée à des codes utilisés au sein du laboratoire et afin d'illustrer la méthode, nous présenterons l'analyse complète d'un exemple (résolution d'un problème elliptique).

La structure ainsi construite a pour but de pouvoir être utilisée facilement au sein d'une équipe de travail, ce qui entraîne d'une part un découpage très clair des classes en fonction des domaines

[1] Laboratoire de Mathématique, Analyse Numérique et EDP, Université Paris Sud
[2] CEA
[3] Doctorant Laboratoire de Mathématique, Analyse Numérique et EDP, Université Paris Sud

de compétences mais aussi une définition précise des classes génériques représentant les objets mathématiques manipulés.

La méthodologie que nous présentons dans cet ouvrage est basée sur un découpage net des classes par compétences. Trois niveaux hiérarchiques sont à distinguer : le premier gérant la définition formel du problème mathématique, le deuxième la description de la méthode de discrétisation employée et enfin le troisième gérant la résolution algébrique du problème traité. Verticalement à cette structure sont superposés quatre blocs distincts : le problème, les opérateurs, les variables, et enfin les domaines de calcul. Il est à noter que sur les niveaux discret et algébrique, un cinquième bloc est nécessaire décrivant pour l'un la méthode de discrétisation et pour l'autre la méthode de stockage. Nous fournissons aux programmeurs des paquetages de classes génériques leur permettant de construire leurs propres hiérarchies, spécifiques au problème qu'ils veulent traiter, mais aussi des paquetages de communication fournis avec des outils de visualisation et contrôle de code.

Ce document est composé de deux parties. La première décrit la démarche méthodologique qui permet de faire le lien entre l'analyse mathématique et la conception informatique du projet. Après avoir détaillé les motivations qui nous ont amenés à concevoir ce projet et présenté un bref état de l'art de ce qui existe, nous décrivons une formalisation des problèmes mathématiques et de leur discrétisation compatible avec une vision orientée objet. Ensuite, un travail de conceptualisation est également effectué au niveau de la compréhension de la notion d'environnement. La lecture de ce chapitre peut également être vu comme une introduction à la méthodologie de conception orienté objet UP (Unified Process).

Dans la seconde partie, plus technique, après avoir décrit la mise en œuvre de la méthode sur un cas d'école (résolution d'un problème elliptique), nous donnons un mode d'emploi des bibliothèques de l'environnement de travail (serveur, instrumentation de code et exploitation graphique), ainsi que le standart de documentation choisi et les outils utilisés.

On pourra également trouver dans les annexes un état de l'art détaillé, un récapitulatif des notations UML (Unified Modeling Language) utilisés, ainsi qu'une introduction à CORBA (Common Object Request Broker Architecture).

Première partie

De l'analyse mathématique à la conception

Chapitre 1

Philosophie et motivations

1.1 Environnement de travail : Recherche et enseignement

L'équipe d'Analyse Numérique et Equations aux Dérivées Partielles du Laboratoire de Mathématique de l'Université de Paris Sud (Orsay) a deux axes d'action aboutissant à l'usage intensitif de l'Informatique Scientifique : *la recherche* et *l'enseignement*. Nous sommes donc tenus, par nature, être à la pointe de la technologie et si possible au delà de façon à enseigner les techniques informatiques ainsi que les algorithmes et les méthodes numériques les plus performants.

Du point de vue de la recherche, notre équipe développe (en particulier) de nouvelles méthodes numériques pour la résolution de problèmes de plus en plus compliqués issus de domaines scientifiques divers. La complexité des problèmes traités fait que les techniques de résolution sont à leur tour d'une difficulté de mise en œuvre telle que les développements informatiques les plus récents deviennent indispensables à leur gestion. De plus la masse des calculs nécessaires à la résolution numérique de tels problèmes nécessite l'utilisation de calculateurs particulièrement puissants.

Par ailleurs, l'environnement universitaire nous permet d'avoir des liens privilégiés avec d'autres communautés scientifiques dans des domaines d'application très variés ce qui nous donnent accès aux motivations physiques des problèmes que nous souhaitons étudier mathématiquement. Les informaticiens nous aident également dans le choix des outils et des langages les plus adaptés nos études.

Du point de vue de l'enseignement, nous avons besoin d'un outil pédagogique simple, afin de proposer aux étudiants une base de travail qu'ils feront évoluer au cours de leur cursus pour arriver au développement d'un code efficace, parallèle, multi-plateformes par exemple. La rapidité de développement est l'un des points important dans le choix d'une méthodologie.

L'environnement informatique de notre équipe a une influence sur les choix de méthodologie de programmation que nous sommes menés à faire. Du point de vue des machines de calcul, l'éducation et la recherche ont toujours plus ou moins utilisé trois niveaux de matériels informatiques :

1. des moyens locaux,
2. des moyens de méso-informatiques au niveau de l'Université,
3. des moyens nationaux.

Nous n'avons bien évidemment pas la prétention de tout réécrire. Nous présentons donc ici une analyse de l'état de l'art dans les domaines des méthodologies de programmation et des bibliothèques et dans celui des logiciels existants dans les matières qui sont les nôtres. Bien sûr, cette analyse ne peut être que partielle et sera complétée au cours du temps.

Un certain nombre de critères tels que l'efficacité, la réutisabilité, le portage, ... sont évidents. Mais dans le cadre de la recherche et de l'enseignement, l'instrumentation des codes est fondamentale. Nous avons besoin de comprendre ou de montrer le comportement des méthodes de calculs afin de pouvoir effectuer des analyses pertinentes.

Dans ce chapitre, nous développons l'ensemble de ces points pour aboutir à un cahier des charges de ce que devrait être notre méthodologie de programmation.

1.1.1 La recherche

Les activités de recherche de l'équipe d'Analyse Numérique et E.D.P. peuvent être regroupées brièvement en trois domaines de compétences.

La recherche fondamentale : c'est l'aspect mathématique du problème qui est ici analysé. Les questions de régularité, d'existence, d'unicité des solutions, de validité du problème sont étudiées. Des analyses plus théoriques sont également menées par nos spécialistes des E.D.P..

La recherche appliquée : le premier point résolu, la question de la méthode de résolution peut être posée, elle conduira à la mise au point de nouveaux algorithmes.

Le calcul scientifique : enfin, la mise en œuvre doit être faite afin de vérifier l'efficacité des méthodes mises au point. Il est également courant que l'étude numérique d'un problème aide dans la compréhension des équations et oriente les études théoriques.

Cela implique que ces trois niveaux soient représentés dans les codes de recherche de l'équipe : théorie des équations, méthodes de résolution et moyens informatiques. Chacun doit travailler à son niveau sans remettre en cause les autres parties des codes de calcul. Le scientifique doit pouvoir résoudre son problème en fonction de ces paramètres. Ainsi, par exemple, la mise au point d'une méthode de résolution ne doit pas interférer avec les techniques d'algèbre linéaire.

Bien que les équations et les techniques de résolution soient différentes d'un problème à l'autre, de nombreux points restent communs à l'ensemble des études : les maillages, l'utilisation de techniques de discrétisations (éléments finis, différences finis, ...), les algorithmes de résolution linéaires et non linéaires, ... Il est donc tout à fait souhaitable de capitaliser les développements et de rendre compatibles les différents logiciels développés au sein de l'équipe. Les avantages sont bien connus et multiples : efficacité, vitesse de développement, fiabilité, réutilisabilité, modularité, ...

La vocation première de ces codes de calcul est la recherche. Cela ne doit pas être en opposition avec certains critères plus souvent appliqués aux codes industriels : documentation, autovérification des données, efficacité, mais qui sont essentiels pour satisfaire nos contraintes (travail collaboratif, capitalisation des développements, ...). Nos codes de calcul doivent ainsi être publiables en satisfaisant au mieux les critères des codes industriels.

1.1.2 L'enseignement

Le calcul scientifique est enseigné dans tous les cycles du cursus universitaire, avec des finalités différentes. Les premiers cycles utilisent l'outil informatique à travers des progiciels afin d'illustrer les propriétés de résultats mathématiques (Matlab, Maple, ...). Puis le besoin de développer ses propres logiciels pour résoudre plus efficacement certains problèmes se fait sentir. On apprend alors des langages de programmation sophistiqués tels que Fortran-95, C++. Ces langages très riches augmentent la difficulté de mise au point des codes de calcul volumineux en respectant des règles de fiabilité et d'efficacité. Il est donc nécessaire d'enseigner et de promouvoir des techniques de programmation avancées. Les étudiants numériciens, qui développeront les codes de calcul de demain, doivent comprendre les mathématiques, connaître les méthodes de résolution ainsi que l'informatique.

Nous sommes confrontés à une problèmatique proche de celle évoquée précédemment pour la recherche, c'est-à-dire de donner aux doctorants un mode de programmation, des outils de bases, ... pour que les codes de calcul soient développés rapidement tout en restant portables, fiables, facilement adaptables, bien documentés. Les étudiants devant avant tout pouvoir rapidement s'intéresser aux problèmes numériques et non se focaliser sur le développement (à moins que ce ne soit le sujet de l'étude).

Comme pour la recherche, il est nécessaire que l'informatique reflète les différents enseignements qui vont de l'illustration de propriétés mathématiques à la conception et à la mise en œuvre de codes de calcul. Chaque enseignant doit ainsi pouvoir agir dans le domaine de compétence qui l'intéresse en ignorant le reste. Cela rejoint tout à fait les éléments d'analyse avancés pour la recherche.

L'apprentissage des langages d'aujourd'hui nécessite des étapes plus nombreuses. Le Fortran-77 était un langage simple, l'analyse procédurale donnait une méthodologie relativement naturelle : on avait un ensemble simple et concis rapidement maîtrisable. La situation actuelle est bien différente avec des langages comme C++, JAVA ou encore FORTRAN-95 qui prennent toute leur mesure dans une conception objet des codes de calcul (même si FORTRAN-95 n'est pas vraiment un langage objet, FORTRAN-2000 le sera). Il n'est pas raisonnable de mettre sur le marché du travail (aussi bien dans la recherche que dans l'industrie) des étudiants ne connaissant pas le concept d'objet. Cette dernière remarque aura une influence prépondérante sur notre cahier des charges.

Si nous devons prendre en compte la programmation objet, il devient donc important de proposer une méthodologie de travail illustrée par de nombreux exemples plus ou moins sophistiqués afin les étudiants utilisent ceux-ci pour assimiler pas à pas les rouages complexes de ces langages mais aussi l'analyse objet.

L'adoption par les étudiants d'une méthodologie de travail a d'autres avantages. L'enseignant peux ainsi plus facilement retrouver l'essentiel dans les travaux de ses étudiants.

Les étudiants que nous formons deviendront à leur tour enseignants ou ingénieurs. Notre ambition est de leur donner une formation aussi complète que possible dans le domaine théorique comme dans le domaine applicatif. En effet, il nous parait souhaitable, voir indispensable, que même les plus théoriciens puissent illustrer leurs propos informatiquement. Enfin, pour les étudiants qui se destinent à une carrière industrielle, il est nécessaire qu'ils aient la maîtrise les outils d'aujourd'hui utilisés par l'industrie.

1.1.3 L'environnement informatique

Comme nous l'avons déjà signalé, nous disposons de plusieurs ressources informatiques à travers les moyens de l'équipe, le centre de méso-informatique de l'Université et les centres nationaux (IDRIS et CINES). Le rôle de chacune de ces trois structures au niveau de la puissance du matériel informatique est différent mais surtout complémentaire. L'accessibilité est également differente, de l'accessibilité en continu des deux premiers niveaux à la demande préalable de moyens sur les centres nationaux.

Au niveau local, nous avons toujours souhaité mettre à la disposition de nos utilisateurs des outils de développement dont les architectures sont les reflets des principales gammes de machines existantes. Dans ce contexte nous avions acquis une machine vectorielle multiprocesseurs à l'époque du CCVR et de ses machines CRAY. Ce calculateur nous a permis de vectoriser nos applications. Aujourd'hui nous disposons d'une grappe de PC biprocesseurs reliés par un réseau rapide. Bien entendu, nous travaillons aussi sur des stations de développement (Sun, SGI, PC, Mac), des stations graphiques, des stations de bureautique et nous utilisons une station de montage vidéo, le tout étant relié par un réseau de classe 5. Nous insistons sur le fait que l'ensemble des moyens de calcul locaux ne représente qu'un outil de développement et ne permet pas d'effectuer des calculs de taille réaliste.

C'est pourquoi le Centre d'Informatique Scientifique du C.R.I. de l'Université de Paris-Sud représente un élément indispensable à notre activité numérique. Le C.I.S. nous offre des machines de puissance moyenne et un environnement de travail plus proche de celui de l'industrie. Ceci nous permet de tester nos codes pour des tailles de problèmes plus réalistes (des tailles réelles dans certains cas) et dans un environnement différent. On y affine aussi le développement en fonction des difficultés apparaissant inévitablement avec l'augmentation de la taille des calculs.

Enfin les centres nationaux permettent l'exploitation réelle de nos codes de calcul grâce à la puissance et la taille mémoire de leurs machines.

Une première conclusion s'impose : nous travaillons dans un environnement très hétérogène. Ceci implique le développement de code **portable** c'est-à-dire qu'il est nécessaire :

- de respecter les normes des langages en ignorant les extensions des constructeurs (quelquefois malheureusement bien utiles) ;
- de transporter l'ensemble de notre environnement de travail, c'est-à-dire d'inclure dans notre environnement des outils comme le "profiling" et autres outils de mesure et d'observation du

comportement des codes afin d'avoir une homogénéité des résultats d'analyse ;
- de prévoir des outils de développements (génération de dépendance pour la compilation, interfaces inter-langages ...) ;
- de prévoir des outils de couplage de codes afin d'avoir un usage optimal du matériel informatique.

1.1.4 En conclusion

En conclusion, pour répondre au mieux à l'ensemble des idées qui ont été données précédemment, il nous faut mettre en œuvre une technique de programmation et un ensemble d'outils et de bibliothèques répondant aux caractéristiques suivantes :
- une facilité de reprises des codes développés,
- l'automatisation de la documentation,
- l'usage des langages les plus adaptés,
- une rationalisation des moyens informatiques (Interface Graphique séparée du code), prenant en compte l'hétérogènéité du parc informatique,
- la prise en compte des notions d'évolutabilité, d'efficacité, de modularité, de réutilisabilité, d'optimisation et de structuration de la programmation,
- la réutilisation des bibliothèques et des logiciels existants,
- la facilité du travail collaboratif entre spécialistes de compétences différentes,
- la capitalisation des acquis.

De plus, nous avons déjà signalé que l'usage de la conception objet était indispensable du point de vue de l'enseignement.

1.2 Les objectifs et leurs contraintes

Notre réflexion actuelle porte sur un ensemble d'objectifs assujettis à un certain nombre de contraintes pour obtenir une méthodologie de travail et donc de programmation.

Comme nous l'avons déjà signalé, les domaines de compétences de notre équipe vont de l'analyse mathématique du problème à l'implémentation sur un ensemble de plateformes données en passant par une recherche sur l'algorithmique de résolution.

Ces trois points de vue sont indépendants et ne sont pas développés par les mêmes personnes. Ceci implique de prévoir des étapes intermédiaires indépendantes afin de profiter des compétences de chacun dans son domaine pour le développement de chacune ces étapes.

Par ailleurs nous voulons une certaine capitalisation du savoir faire. Il est en effet indispensable de pouvoir réutiliser un code de calcul alors que le dévelopeur n'est pas disponible ou d'utiliser une application ancienne pour en construire une nouvelle. Il s'agit donc de disposer d'une bibliothèque fiable permettant le dévelopement de nouvelles applications rapidement.

Les développements doivent être homogènes, collaboratifs et normalisés afin de permettre à plusieurs personnes de travailler sur les mêmes projets.

Enfin nous avons déjà signalé que la maîtrise de la conception orientée objet était indispensable aussi bien pour les futurs enseignants que pour les futures ingénieurs.

Le sous-paragraphe 1.1.4 résume déjà un certain nombre de nos objectifs et de nos contraintes. En complément de ces points déjà évoqués, on peut ajouter les exigences suivantes :
- L'utilisateur final doit pouvoir travailler sans recompiler. Il doit donc avoir accès facilement à l'ensemble des paramètres physiques de son problème et disposer des éléments nécessaires à l'appréciation du comportement du code de calcul
- L'une des directions de recherche importante dans notre équipe est la mise au point de méthodes de résolution nouvelles. Il est donc important de pouvoir agir sur ces méthodes indépendamment du reste du code. Ces méthodes peuvent être couplées ou non avec les techniques de discrétisations.
- Nous souhaitons une certaine indépendances des codes de calcul vis à vis des méthodes de discrétisation. Dans un cas idéal nous voudrions pouvoir remplacer différences finis par

éléments finis (par exemple) sans modifier le reste du code. S. Labbé (à paraître) donne une description unifiée des principales méthodes de discrétisation (voir également 2). Notre mise en œuvre doit pouvoir intégrer ce formalisme.
- Enfin, les méthodes de résolution linéaire et les techniques de stockage associées forment un ensemble à part. Des bibliothèques externes sont utilisées de façon courantes pour ces opérations. Cet ensemble de bas niveau doit avant tout rester très efficace. Mais il doit également pourvoir être interchangable.

Les objectifs et les contraintes que nous venons de fixer nous permettent ainsi de cadrer notre recherche par rapport aux outils et bibliothèques existants dont le paragraphe suivant présente un aperçu non exhaustif.

1.3 L'état de l'art en bibliothèques et en outils de calcul scientifique

Ce paragraphe représente un résumé de l'annexe A. L'ensemble des bibliothèques et des outils de calcul scientifique est immense. Il n'est pas question ici de faire le tour complet de cet ensemble. Il est donc évident que le contenu de ce paragraphe et de l'annexe est partiel et partial. Nous avons décidé de ne considérer que les outils du domaine publique dont les sources sont disponibles. Les raisons de ce choix très restrictif sont évidemment financières mais elles portent aussi sur des questions de disponibilité et de portabilité des outils quelque soit l'environnement de travail (puisqu'il suffit de recompiler). Par ailleurs, il sera probablement nécessaire d'interfacer ces outils pour les rendre compatibles avec nos outils. Le fait d'avoir accès aux sources facilite grandement cet interfaçage.

Il y a deux parties disjointes dans notre revue des outils de calcul scientifique. La première concerne les outils spécialisés intervenant à un endroit particulier d'une méthodologie globale de développement. La résolution d'un système linéaire en est un exemple typique. La seconde concerne les travaux qui proposent une méthodologie et des outils pouvant peut-être répondre à notre cahier des charges défini dans le paragraphe précédent. Par contre, ces développements correspondent aux besoins et contraintes formulés par leurs concepteurs qui peuvent être différents des nôtres.

Il est clair que nous n'avons pas la prétention de tout réécrire notamment dans les domaines de l'algèbre linéaire des méthodes directes, des mailleurs 2D-3D et des applications graphiques en particulier. Les trois exemples précédents représentent trois domaines de compétences particuliers pour lesquels il existe de très nombreux travaux particulièrement performants. Reprogrammer tout ce savoir faire est clairement une perte de temps voir même impossible quand à l'obtention d'une efficacité équivalente.

Pour le premier, la bibliothèque LAPACK [**?**] écrit en Fortran-77 et ces variantes, CLAPACK, LAPACK++ [**?**], LAPACK90 et ScaLapack sont incontournables et règlent le problème de l'algèbre linéaire des méthodes directes. L'algèbre linéaire itérative peut être réécrite. Elle est effectivement plus simple à programmer.

La plupart des applications sur lesquelles nous travaillons utilisent la discrétisation d'un domaine de calcul qui est, en général un ouvert borné de R^2 ou de R^3. Il existe peu de développements de mailleurs 2D-3D du domaine public. Le plus accessible est celui de MODULEF. Il faut aussi signaler le projet GAMMA : Génération Automatique de Maillages et Méthodes d'Adaptation de l'INRIA qui offre quelques mailleurs de très bonne qualité.

Concernant les logiciels graphiques, nous utilisons depuis longtemps le logiciel commercial AVS/Express. Concernant les logiciels un domaine public, on peut citer open-DX et VTK, ainsi que certaines extensions graphique de JAVA.

De façon générale, il faut utiliser les logiciels et bibliothèques spécifiques lorsqu'ils sont nécessaires.

Il faut également noter les travaux plus globaux comme `elsa` développé à l'ONERA qui est un logiciel de simulation en Aérodynamique, `FreeFEM++` qui résoud des EDP par éléments finis, `Overture` développé au LLNL (Lawrence Livermore National Laboratory) qui offre un ensemble

de classes pour la résolution de problèmes aux différences finies et volumes finis. Ces travaux ont leurs finalités propres avec des contraintes spécifiques (différentes des notres) (voir l'annexe A).

1.4 Choix de la méthodologie

Notre environnement de travail, nos besoins et les contraintes que nous devons prendre en compte nous ont conduits à élaborer un cahier de charges précis. A partir de ces données, nous avons cherché d'abord à évaluer les logiciels et bibliothèques existants.

Certains de ces développements sont incontournables, mais aucune approche globale ne semble répondre à tous les points de notre cahier des charges.

Il nous faut donc envisager la mise au point d'une méthodologie de travail et de développements correspondante plus spécifiquement à nos attentes.

Comme nous l'avons déjà signalé, nous devons avoir une conception objet pour notre modèle de code de calcul scientifique. Grâce à ce point de vue, nous disposons de l'ensemble des formalismes objet comme

- le processus unifié (unified process),
- les patrons architecturaux (design patterns).

D'autre part, nous avons vu la nécessité de réutiliser certaines des bibliothèques existantes. Cette contrainte supplémentaire va conduire à certains choix techniques : ainsi, il faudra par exemple assurer la compatibilité de la notion de tableau avec l'ensemble LAPACK, interfacer certaines es librairies graphiques ....

Nous n'avons pas encore vraiment parlé de langages de programmation, c'est bien entendu volontaire. L'essentiel de notre étude est la mise au point d'une méthodologie de conception de code de calcul scientifique, elle doit donc être indépendante de tout langage. Une fois cette méthodologie mise au point, il sera alors aisé de l'adapter en fonction des possibilités des langages. Nous pensons prendre en compte les langages C++, Fortran-95 et JAVA. Pour l'instrumentation et le couplage de codes nous ajouterons le langage C.

Chapitre 2

Analyse et conception objet du problème mathématique

Introduction

La résolution numérique d'un problème de calcul scientifique constitué d'un système d'équations différentielles s'effectue en plusieurs étapes qui permettent de passer du problème continu aux systèmes algébriques à inverser. Ces différentes phases, concernant chacune un domaine de compétence particulier, ont été rappelées dans le chapitre précédent :

- la théorie des équations,
- les méthodes de résolution,
- la mise en œuvre informatique.

Chacune de ces étapes correspond à un cadre mathématique particulier, pour lequel est défini un certain nombre d'entités (espace, opérateur, ...). Pour analyser ces différentes phases, nous nous baserons sur l'étude effectuée sur la discrétisation des équations différentielles (à paraître). Le principe exposé consiste à conceptualiser ces entités mathématiques dans un esprit d'implémentation informatique. Ce cheminement nous conduira à la notion d'objets mathématiques, qui feront le lien entre l'abstraction théorique et l'implémentation technique. Ainsi, les différentes étapes de la résolution décrites dans l'étude théorique se retrouvent dans le découpage objet des composantes logicielles des codes correspondants. La structure de base des composantes mathématiques sera donc divisée en trois couches représentant les différents niveaux d'abstraction de la résolution :

- le niveau mathématique qui définit de façon abstraite le problème à traiter, correspondant à la première phase de la résolution.
- Le niveau de discrétisation décrivant la méthode de discrétisation du problème, correspondant à la deuxième phase de la résolution.
- Le niveau algébrique correspondant à la troisième phase de la résolution, c'est-à-dire aux calculs proprement dits.

Ces trois niveaux successifs permettent de résoudre le problème mathématique initial dans sa globalité. Cette résolution nécessite des interfaces de communication avec l'utilisateur lui permettant d'analyser la ou les phases du processus qui l'intéressent. Cette structure transverse correspond donc d'une façon générale à la gestion des communications entre les abstractions mathématiques et l'extérieur. Ce niveau supplémentaire, non mathématique, sera nommé utilitaire.

Dans un premier temps, nous décrivons le cadre théorique sur lequel est basé la notion d'objet mathématique. A partir de cette structure, nous pouvons alors définir les différentes entités mathématiques intervenant dans chacun des niveaux décrits ci-dessus. Enfin, la concrétisation de ces abstractions mathématiques nous conduit à la conception détaillée de ces différents niveaux.

2.1 Cadre théorique

L'objectif du projet est de résoudre de façon générique, c'est-à-dire quelle que soit la méthode de discrétisation, un système d'équations aux dérivées partielles.Dans ce paragraphe, nous décrivons une représentation unifiée des méthodes de discrétisation usuelles (éléments finis, différences finies et volumes finis). Celle-ci repose sur des opérateurs de plongement et de restriction d'espaces fonctionnels vers des espaces de dimension finie.

Supposons donc que l'on veuille traiter le problème suivant :

$$\text{soit} f \in \mathrm{W}_2, \text{ trouver } u \in \mathrm{W}_1 \text{ tel que } \mathcal{P}u = f, \tag{2.1}$$

où W_1 et W_2 sont deux espaces fonctionnels (de manière trés générale, on peut par exemple dire que ce sont des espaces de Banach séparables), $\mathcal{P}$ représente un opérateur linéaire ou non, u la solution, et f les forces extérieures. Afin de résoudre ce système, il est nécessaire de définir un problème approché c'est-à-dire d'écrire un système approché dans des espaces de dimension finie sous la forme

$$\text{soit} f_h \in \mathrm{W}_{2,h}, \text{ trouver } u_h \in \mathrm{W}_{1,h} \text{ tel que } \mathcal{P}_h u_h = f_h, \tag{2.2}$$

où $\mathrm{W}_{1,h}$ et $\mathrm{W}_{2,h}$ sont deux espaces de dimension finie, u_h la solution approchée et f_h l'approximation des forces extérieures. $\mathcal{P}_h$ est une approximation de $\mathcal{P}$ à laquelle nous allons donner un sens.

Considérons les opérateurs de plongement $\mathrm{P}_h^* : \mathrm{W}_{1,h} \mapsto \mathrm{W}_1$ (resp. $\mathrm{Q}_h^* : \mathrm{W}_{2,h} \mapsto \mathrm{W}_2$) et de restriction $\mathrm{P}_h : \mathrm{W}_1 \mapsto \mathrm{W}_{1,h}$ (resp. $\mathrm{Q}_h : \mathrm{W}_2 \mapsto \mathrm{W}_{2,h}$). On obtient alors le diagramme suivant :

$$\begin{array}{ccc}
\mathrm{W}_{1,h} & \overset{\mathcal{P}_h}{} & \mathrm{W}_{2,h} \\
\mathrm{P}_h \downarrow & & \uparrow \mathrm{Q}_h^* \\
\mathrm{W}_1 & \overset{\mathcal{P}}{} & \mathrm{W}_2 \\
\mathrm{P}_h^* \uparrow & & \downarrow \mathrm{Q}_h \\
\mathrm{W}_{1,h} & \overset{\mathcal{P}_h}{} & \mathrm{W}_{2,h}
\end{array}$$

On peut ainsi définir l'opérateur discret $\mathcal{P}_h$ en fonction des opérateurs de restriction et de plongement et du problème continu $\mathcal{P}$:

$$\forall h \in \mathbb{R}^p, h > 0, \mathcal{P}_h = \mathrm{Q}_h \circ \mathcal{P} \circ \mathrm{P}_h^*.$$

Ces opérateurs de discrétisation sont construits de façon à vérifier un certain nombre d'hypothèses permettant d'assurer qu'ils sont candidats pour être de "bonnes" discrétisations.

Hypothèse 2.1.1 *On suppose que les opérateurs P_h, Q_h, P_h^*, Q_h^* sont tels que :*
(i) Pour tout h dans $\mathbb{R}^p$, P_h et Q_h sont des opérateurs linéaires continus.
(ii) On a :

$$\begin{aligned}
\forall u \in W_1, \lim_{|h|\to 0} ||P_h^* \circ P_h \; u - u||_{W_1} = 0, \\
\forall u \in W_2, \lim_{|h|\to 0} ||Q_h^* \circ Q_h \; u - u||_{W_2} = 0, \\
\forall u \in W_{1,h}, \lim_{|h|\to 0} ||P_h \circ P_h^* \; u - u||_{W_{1,h}} = 0, \\
\forall u \in W_{2,h}, \lim_{|h|\to 0} ||Q_h \circ Q_h^* \; u - u||_{W_{2,h}} = 0.
\end{aligned}$$

Dans le cas, fréquent, où $W_{1,h}$ est inclus dans W_1 et $W_{2,h}$ est inclus dans W_2, les deux dernières lignes du point *(ii)* de l'hypothèse (2.1.1) sont automatiquement vérifiées.

On démontre alors que les discrétisations ainsi construites sont automatiquement consistantes, la stabilité quant à elle dépend fortement, bien entendu, du problème continu.

Ainsi, afin d'illustrer ceci, nous montrons ici deux exemples d'opérateurs : les éléments finis P1 et les différences finies.

En éléments finis, l'espace $W_{1,h}$ sera celui engendré par les fonctions de base $(\varphi_i)_{i=1,..,N}$, où N est le nombre de sommets de la triangulation (on notera pour la suite que les sommets numérotés entre 0 et N_{int} sont à l'intérieur du maillage et les autres sur le bord du maillage). Dans notre exemple, les fonctions de base sont de type P1, tandis que l'espace $W_{2,h}$ est celui des fonctions constantes par morceaux sur les cellules du maillage. Ainsi, les opérateurs P_h et P_h^* sont définis comme suit :

$$\forall u \in W_1,\ P_h(u) = [M^{-1}(\Phi, u)].\Phi,$$
$$\forall u_h \in W_{1,h},\ P_h^*(u_h) = u_h,$$

où $(.,.)$ désigne le produit scalaire dans L^2, $\Phi = (\varphi_1, ..., \varphi_N)^t$ et $M = (\Phi, \Phi^t)$.

En ce qui concerne la méthode des différences finies, on adopte ici une technique de collocation qui est présentée, par soucis de "légèreté" de l'écriture, sur le maillage régulier du carré $[0,1]\times[0,1]$ dont les noeuds sont les points $x_i = (p\ h, l\ h)^t = (x_{i,1}, x_{i,2})^t$ pour $i = (p,l)^t$ et p et l variant entre 0 et N $(h = 1/N)$; de plus, on pose $\omega_i = [(p-1/2)h, (p+1/2)h]\times[(l-1/2)h, (l+1/2)h]\cap[0,1]\times[0,1]$. Considérons la notation suivante

$$\forall i \in \{0, ..., N\} \times \{0, ..., N\},$$
$$P_i(x) = ((x_1 - x_{i,1})^2, (x_2 - x_{i,2})^2, (x_1 - x_{i,1}), (x_2 - x_{i,2}), 1)^t.$$

Alors on montre qu'il existe une matrice inversible A carrée d'ordre 5 (qui ne dépend pas du point dans le cas exposé ici des maillages réguliers), telle que, pour tout élément u de W_1, $P_h u$ coïncide avec u aux points du maillage. L'opérateur $P_h u$ s'exprime sous la forme :

$$\forall u \in W_1, \forall i \in \{0, ..., N\} \times \{0, ..., N\}, \forall x \in \omega_i,\ (P_h(u))(x) = A^{-1} Y_i . P_i(x),$$

où Y_i est le vecteur des valeurs de u sur la maille i et les quatres mailles voisines. Le relèvement $P_h^* u_h$ d'un élément u_h de $W_{1,h}$ est quant à lui la fonction de W_1 la plus proche u_h au sens de la norme de W_1 et coïncidant avec u_h aux points du maillage.

2.2 Analyse

L'étude théorique précédente nous a permis de définir un certain nombre d'entités mathématiques. Il s'agit maintenant de les faire vivre dans un ensemble cohérent, afin d'aboutir à un modèle d'analyse complet.

Cette phase d'analyse doit permettre de décrire précisemment les éléments qui constitueront la bibliothèque sans indiquer comment les choses seront implémentées.

Il faut dans un premier temps identifier les objets métiers issus du cadre théorique précédent.

Dans l'écriture du problème discret proposée dans la section précédente, nous avons séparé l'opérateur à discrétiser de la discrétisation. Ainsi, $\mathcal{P}$ est défini dans le niveau mathématique tandis que les opérateurs P_h, P_h^*, Q_h et Q_h^* sont définis dans le niveau discrétisation.

Cette remarque nous permet de caractériser deux objets mathématiques particuliers :

- `Operateur` : cette entité correspond à la notion d'opérateur continu illustré dans l'étude théorique par la notation $\mathcal{P}$ (par exemple laplacien, grad, div, dérivée par rapport à x, trace sur le bord ...).
- `Discretisation` : cette abstraction représente les méthode de projection et de relèvement des espaces continus vers les espaces discrets et inversement.

La notion d'opérateur nous conduit de façon naturelle à la notion de variable, une équation correspondant à l'application d'opérateurs à un certain nombre de variables. Nous caractérisons ainsi un nouveau concept :

- `Variable` : cette entité se rapporte aux éléments des différents espaces de dimension infinie manipulés dans le problème à résoudre. Elle correspond par exemple aux solutions des équations, aux conditions initiales, ... Les variables peuvent être de type volumique ou de type surfacique selon qu'elles sont définies à l'intérieur du domaine ou sur le bord.

Enfin, les espaces sont définis sur des domaines, que l'on peut caractériser de la façon suivante :

- `Domaine` : cet objet mathématique représente la topologie continue ou discrète des domaines, en général de $\mathbb{R}^n$, sur lesquels les espaces fonctionnels sont définis.

Les entités précédentes correspondent à tous les éléments intervenant dans un problème mathématique tel qu'il a été défini dans la partie théorique 2.1. Il est ainsi naturel de représenter le problème formel (2.1) en tant qu'entité elle-même, manipulant toutes les autres :

- `Probleme` : cette abstraction correspond à un ensemble cohérent dans lequel vivent et interagissent les autres entités.

Nous venons de conceptualiser les éléments mathématiques qui sont mis en œuvre lors de la définition du problème à résoudre. Il faut maintenant prendre en compte les différentes phases de la résolution. D'un point de vue textuel, on peut les décrire de la façon suivante :

- le **problème formel** est défini sur des espaces continus. Il s'agit d'une ou de plusieurs équations portant sur des **domaines continus**, faisant intervenir des **opérateurs continus** agissant sur des **variables continues**.
- le **problème discret** est obtenu par le choix d'une **méthode de discrétisation** qui permet de discrétiser les opérateurs et variables du problème continu en **opérateurs discrets** et **variables discrètes** et conduit au choix d'un **maillage** du domaine continu.
- les **systèmes linéaires** sont obtenus par assemblages des **matrices** et **vecteurs** issus des opérateurs et variables discrets. Le **stockage** dépend de la forme des matrices. L'inversion de ces systèmes peut être réalisée de manière directe ou itérative.

Le déroulement global d'une résolution peut ainsi être décrit à l'aide des concepts que l'on vient de définir. Ce scénario est illustré par le diagramme de collaboration 2.1.

Comme nous allons le voir en détail dans les paragraphes suivants, chaque couche correspond donc à la résolution du problème à un niveau donné. Chaque niveau repose sur un objet mathématique général, de nom générique problème, manipulant les autres entités du même niveau. C'est pour cette raison qu'ils sont représentés sous forme d'objet contrôle dans le diagramme de collaboration. Comme le diagramme 2.1 le montre, chaque étape de la résolution conduit à la construction du problème de l'étape suivante. Le diagramme de séquence 2.2 permet de rendre compte de la succession des évènements concernant les objets mathématiques de type problème au cours du calcul.

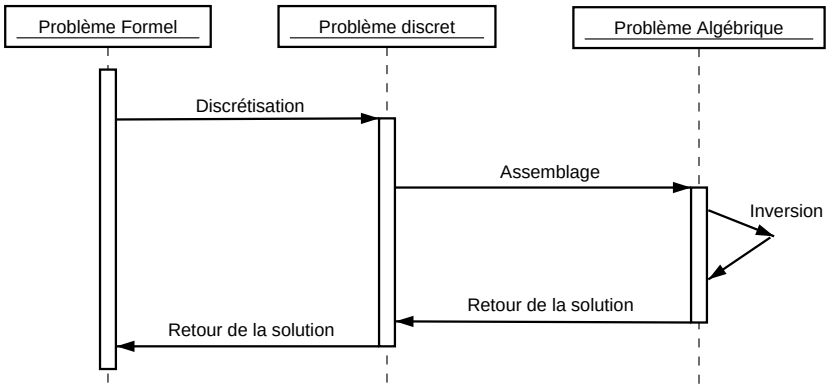

FIG. 2.2 – Diagramme de séquence global représentant la résolution d'un problème mathématique.

Tous ces éléments nous conduisent à l'élaboration d'un modèle d'analyse qui, sous la forme

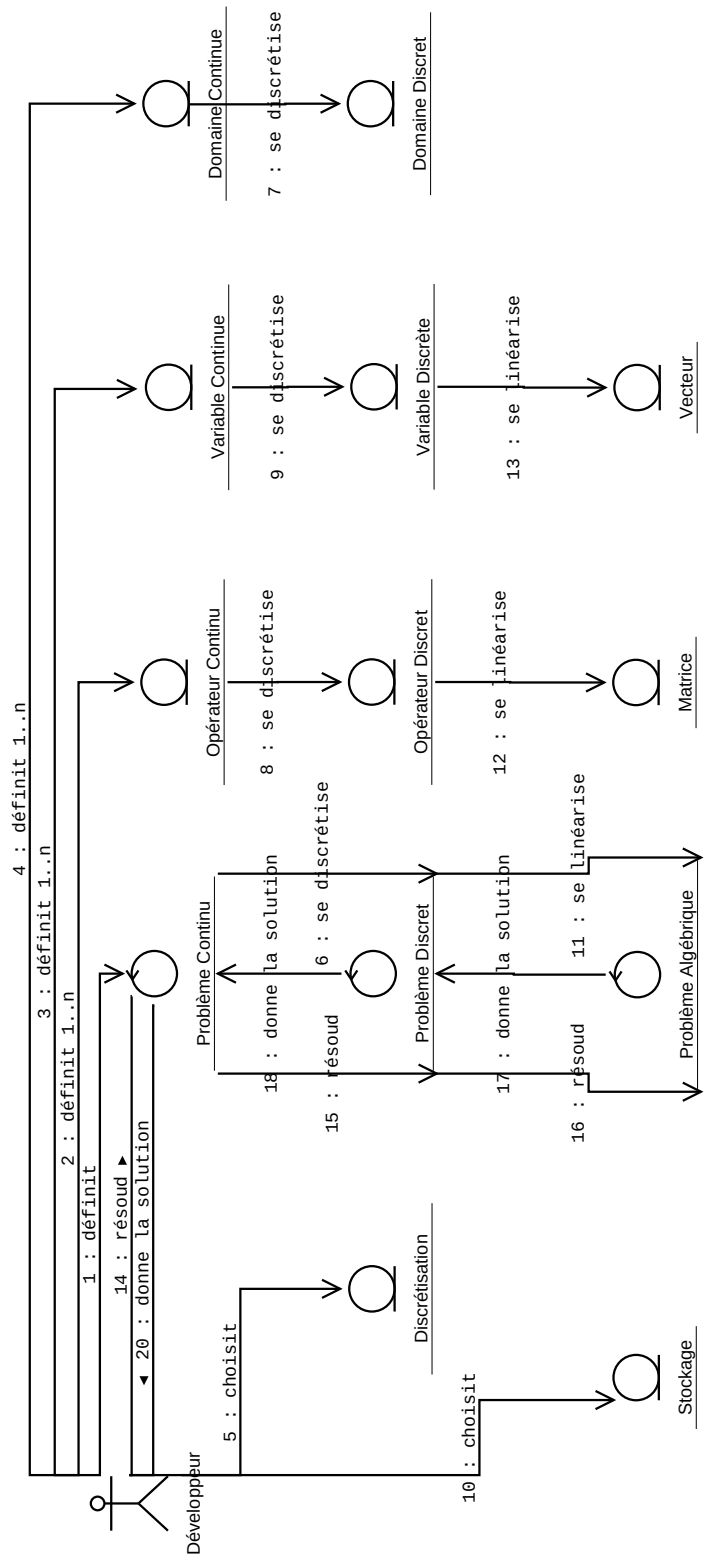

FIG. 2.1 – Diagramme de collaboration global de la résolution.

du diagramme de classe 2.3, met en valeur les différentes classes ainsi que leurs responsabilités respectives.

2.3 Conception préliminaire

Le but de la conception préliminaire est de concevoir des encapsulations métiers.

Nous avons jusqu'à présent travaillé sur un premier découpage logique correspondant aux niveaux d'abstraction des étapes de la résolution du problème. L'étude du modèle d'analyse que nous venons d'obtenir, illustré par le diagramme 2.3, nous conduit à réaliser un deuxième découpage métier, en effectuant des regroupement de classes d'un point de vue sémantique. Il s'agit de rassembler les classes ayant des logiques métier équivalentes. D'une façon générale, elles sont liées par le même substantif. Le diagramme du modèle d'analyse nous amène naturellement à définir les paquetages suivanté :

- le paquetage `Problème` qui réunit les classes `Problème Continu`, `Problème Discret` et `Problème Algébrique`,
- le paquetage `Opérateur` qui regroupe les classes `Opérateur Continu`, `Opérateur Discret` et `Matrice`.
- le paquetage `Variable` qui rassemble les classes `Variable Continue`, `Variable Discrète` et `Vecteur`,
- le paquetage `Domaine`, qui agglomère les classes `Domaine Continu` et `Domaine Discret`.

Les interactions existantes au sein du paquetage `Problème`, illustrée par le diagramme 2.2 se retrouvent de façon analogue dans les autres paquetages. Le diagramme 2.4 montre l'enchaînement des évènements au sein du paquetage `Opérateur`.

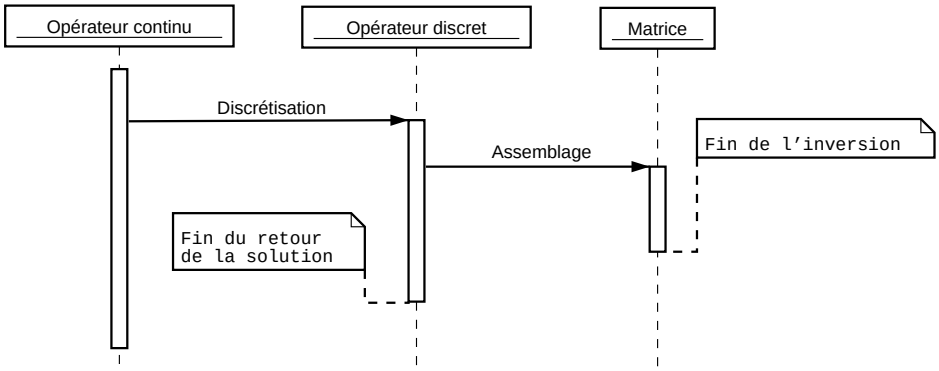

FIG. 2.4 – Diagramme de séquence global représentant la résolution d'un problème mathématique au niveau des opérateurs.

2.4 Conception détaillée

Il faut maintenant finaliser la conception :

- définir définitivement les interactions entres classes,
- préciser de façon détaillée les services attendues pour chacune de ces classes,
- tendre vers des paquetages de plus en plus indépendants pour assurer la souplesse du logiciel.

A ce point du processus, il faut tenir compte de cinq critères fondamentaux (voir [**?**]) essentiels pour s'assurer des bénéfices que peut apporter la programmation objet :

Le couplage : l'interdépendance des classes est un point extrêmement important. Il faut adopter un certain niveau de granularité adapté au problème à traiter. Il est nécessaireé :

- qu'au niveau horizontal, il y ait une réelle souplesse entre les classes de façon à pouvoir passer d'une concrétisation à l'autre sur un paquetage sans avoir à réécrire aucun autre (changer de discrétisation, de domaine, de stockage ...).

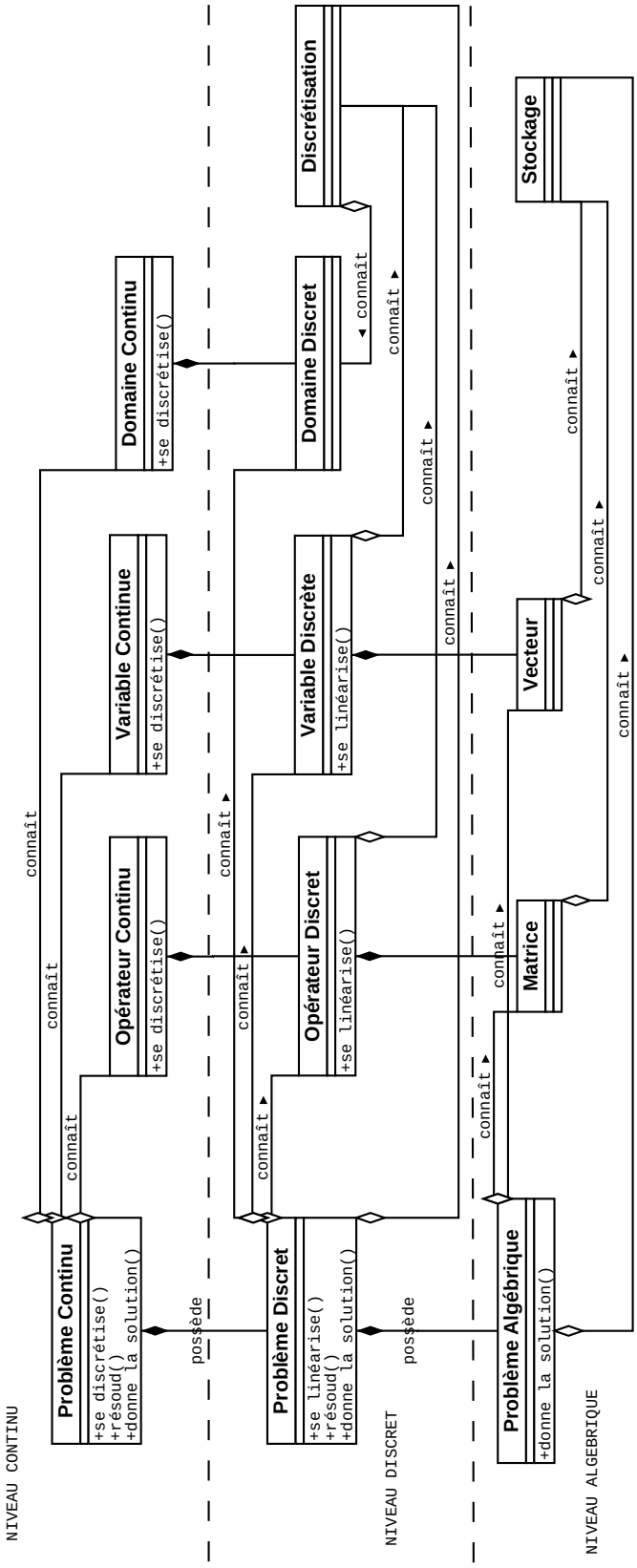

FIG. 2.3 – Modèle d'analyse du problème mathématique.

– qu'au niveau vertical, il y ait une indépendance de réalisation qui permettent à chacun de travailler à son niveau de compétences.

La cohésion : elle mesure le degré de connectivité existant entre les éléments d'une classe unique. Dans notre cas, les classes sont directement issues des entités métiers qu'elles modélisent. Elles sont donc naturellement cohérentes. Ainsi si on considère deux instances de la classe `Discrétisation`, par exemple l'une correspondant aux éléments finis, et l'autre aux différences finies, elles ont effectivement le même comportement (selon le cadre théorique fixé). Seules leurs données internes varient. La cohésion des classes est donc bien respectée.

L'autarcie : l'autosuffisance d'une classe désigne sa faculté d'offrir à elle seule un vrai service, sans nécessiter la collaboration d'autres classes. Compte tenu de la proximité des classes avec les éléments mathématiques qu'elles modélisent, elles en reflètent fidèlement le comportement en respectant ansi cette notion.

La complétude : une classe complète doit fournir tous les services liés au domaine qu'elle est censée modéliser. Il faut donc s'assurer que les services proposés par chacune des classes du modèle correspondent à toutes les manipulations mathématiques réalisables sur les éléments qu'elles représentent, dans le cadre du problème traité.

La primitivité : la notion de primitivité est proche de la notion de couche. La modélisation proposée répond donc particulièrement bien à ce critère : les primitives du niveau n s'appuient sur les primitives du niveau n-1 et offre un certain nombre de services qui sont utilisés par les primitives du niveau n+1.

Compte tenu de ces critères et en s'appuyant sur le modèle d'analyse et le découpage métier de la conception préliminaire, on peut maintenant affiner, détailler et compléter les unités du logiciels.

2.4.1 Conception détaillée du paquetage `Problème`

Le paquetage `Problème` contient les classes `Problème Continu`, `Problème Discret` et `Problème Algébrique`.

La classe `Problème Continu` modélise le problème mathématique 2.1. La classe `Problème Discret` représente le problème mathématique 2.2, approximation de 2.1 basée sur le choix d'une discrétisation. La linéarisation de 2.2 conduit à un système linéaire modélisé par la classe `Problème Algébrique`.

L'étude mathématique des équations nous permet d'affiner la description des classes du modèle.

Pour définir un problème continu, il faut connaître :

– une liste d'opérateurs continus,
– une liste de variables continues, contenant les données du problème (par exemple le second membre `f`),
– une liste de variables continues, contenant les inconnues du problèmes,
– une liste de domaines continus contenant les différents domaines nécessaires à la définition des espaces de travail (domaines volumiques ou surfaciques).

Les services proposés par la classe `Problème Continu` sont de deux types :

– des services de construction du problème continu. Le développeur peut ainsi composer son problème en fonction des équations mathématiques qu'il a à traiter. Ces services lui permettent d'ajouter de nouveaux opérateurs ou de nouvelles données pour définir complétement le problème qui l'intéresse.
– des services de contrôle de la résolution. Une fois le problème assemblé, le développeur peut demander sa résolution. Celle-ci va consister en la discrétisation du problème, c'est-à-dire en la création du `Problème Discret` associé.

L'existence de l'objet `Problème Discret` crée par un objet `Problème Continu` n'a de réalité que pendant la durée de vie de cet objet `Problème Continu`. D'autre part, il ne peut correspondre qu'à cet objet `Problème Continu` particulier. On est donc dans le cas d'une agrégation forte, ou composition.

Par contre, les objets représentant les opérateurs continus, les variables continues et les domaines continus peuvent avoir une existence en dehors de l'objet `Problème Continu` les utilisant.

En effet, l'instanciation de plusieurs `Problème Continu` est parfaitement possible, et les équations mathématiques qu'ils modélisent peuvent très bien comporter un ou plusieurs opérateurs continus ou variables continues identiques. Il faut, dans ce cas, éviter la duplication d'instances équivalentes. On est donc dans le cas d'agrégation ordinaire, dite agrégation faible.

Pour définir un problème discret, il faut construire :
- la liste des opérateurs discrets approchant les opérateurs continus du problème continu correspondant,
- la liste des variables discrètes approchant les données du problème continu correspondant,
- la liste des variables discrètes approchant les inconnues du problème continu correspondant,
- la liste des domaines discrets approchant les domaines continus du problème continu correspondant.

L'obtention de toutes ces entités discrètes nécessite la connaissance de la discrétisation choisie.

La construction du problème discret est géré par le problème continu. Les services proposés par la classe `Problème Discret` sont donc d'un seul typeé :
- des services de contrôle de la résolution, rélisant la deuxième phase de cette résolution, qui consiste en la linéarisation du problème discret, c'est-à-dire en la création du `Problème Algébrique` associé.

Les remarques sur les associations existantes entres les objets du niveau discret au sein de la classe `Problème Discret` sont identiques à celles faites au niveau continu. Ainsi, le cycle de vie de l'objet `Problème Algébrique` est lié au cycle de vie de l'objet `Problème Discret` qui l'a créé. Il s'agit encore d'une composition. Par contre, les objets représentants les opérateurs discrets, les variables discrètes et les domaines discrets sont liés à l'objet `Problème Discret` par agrégation simple pour les mêmes raisons que celles évoquées précédemment.

Enfin, une même discrétisation peut être utilisée pour la résolution de plusieurs problèmes. L'objet `Discrétisation` est donc associé à l'objet `Problème Discret` qui l'utilise par une agrégation faible.

Pour définir un système linéaire, il faut construire :
- la liste des matrices linéarisant les opérateurs discrets du problème discret correspondant,
- la liste des vecteurs linéarisant les données discrètes du problème discret correspondant,
- la liste des vecteurs linéarisant les inconnues discrètes du problème discret correspondant.

La mise en œuvre informatique des matrices et vecteurs nécessite le choix d'une méthode de stockage optimisant l'espace mémoire utilisé et l'accès aux éléments.

Comme pour le niveau précédent, la création du système linéaire est géré par le problème discret. Les services proposés par la classe `Problème Algébrique` sont donc d'un seul type :
- des services de contrôle de la résolution, rélisant la troisième phase de cette résolution, qui consiste en l'inversion du système linéaire, et au calcul de la solution vectorielle.

Les associations entre l'objet `Problème Algébrique` et les objets des classe `Matrice` et `Vecteur` sont de type agrégations faibles, comme pour les niveaux précédents.

L'objet `Stockage` joue au niveau algébrique le même rôle que l'objet `Discrétisation` au niveau discret. Les classes `Problème Algébrique` et `Stockage` sont donc associée par une agrégation simple, pour des raisons analogues à celles évoquées au niveau précédent.

La modélisation UML de la conception détaillée du paquetage `Problème` est illustrée par le diagramme de classes 2.5. Pour des raisons de format des caractères, les noms des classes ont été simplifiés : `Probleme` correspond à `Problème Continu`, `Probleme_h` à `Problème Discret` et `Probleme_Algebrique` à `Problème Algébrique`.

2.4.2 Conception détaillée du paquetage `Opérateur`

Le paquetage `Opérateur` comprend les hiérarchies de classes `Opérateur Continu`, `Opérateur Discret` et `Matrice`.

La classe `Opérateur Continu` manipule peu de données car la plupart des opérations se font dans les classes des niveaux inférieurs. Cependant, un opérateur est nécessairement défini sur un domaine donné.

Ainsi, pour définir un opérateur continu, il faut connaître :

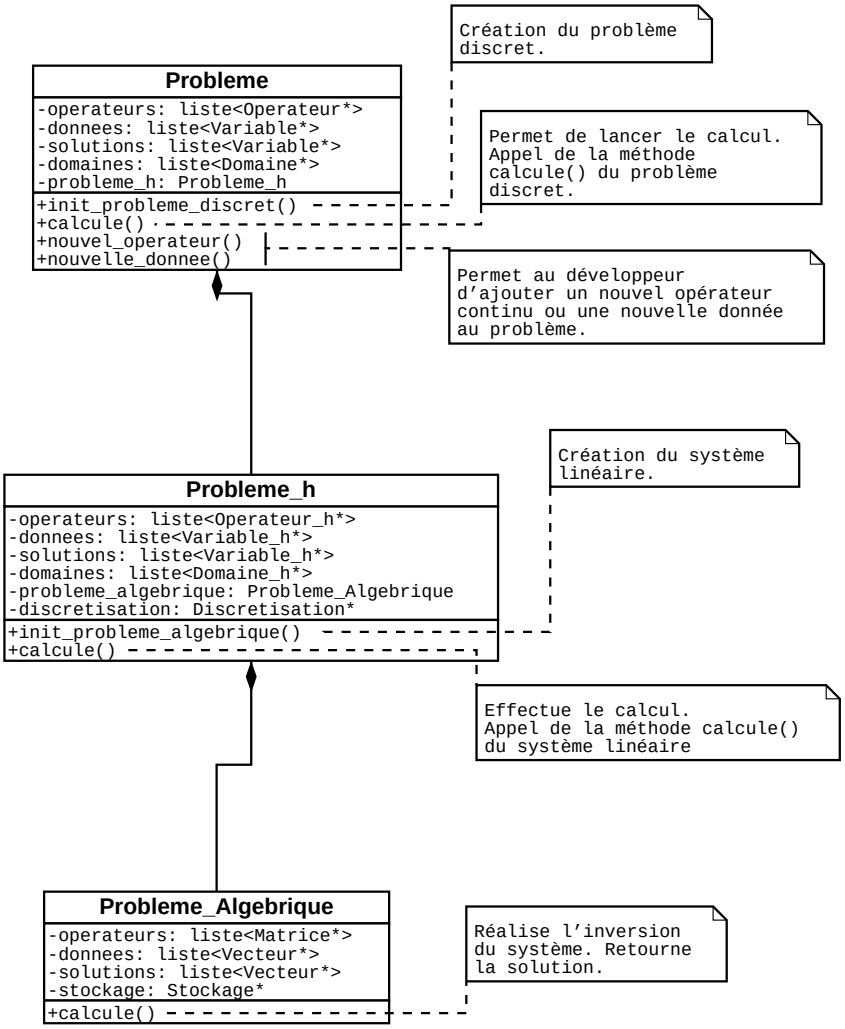

FIG. 2.5 – Diagramme de classes du paquetage `Problème`

.

– une liste de domaines continues sur lesquels est défini l'opérateur. Celui-ci peut en effet être défini sur l'intérieur d'un domaine mathématique, et sur le bord de ce même domaine mathématique. Or le traitement de ces deux cas est susceptible de différer, ce qui conduit à instancier deux domaines continus différents pour traiter informatiquement le problème.

Les services proposées par cette classe ne sont que d'une seule sorte :

– des services de discrétisation de l'opérateur continu, qui vont consister en la création de l'objet `Opérateur Discret` associé.

Le cycle de vie des objets du paquetage `Opérateur` est similaire à celui présenté pour le paquetage `Problème`. Ainsi, l'existence de l'objet `Opérateur Discret` n'a de réalité que pendant la durée de vie de l'objet `Opérateur Continu` qui l'a créé, d'ou une association de type composition entre ces classes. Inversement, les domaines continus sont liés à l'objet `Opérateur Continu` par agrégation simple, car leurs existences ne sont pas assujetties à celle de cet objet.

Pour définir un opérateur discret, il résulte de l'analyse précédente qu'il faut connaîtreé :

– la liste des domaines discret correspondants aux domaines continus sur lesquels est défini l'opérateur continu.

Enfin, la discrétisation de l'opérateur nécessite évidemment de connaître la méthode de discrétisation choisie.

Comme pour sa primitive continue, les services proposés par la classe `Opérateur Discret` ne sont que d'un seul ordre :

– des services de linéarisation de l'opérateur discret, qui vont consister en la création de l'objet `Matrice` associé.

On retrouve les mêmes associations entre classes que celles explicitées précédemmenté :

– association de type composition avec la classe `Matrice`,
– association de type agrégation simple avec les classes des domaines discrets,
– association de type agrégation simple avec la classe `Discrétisation`.

La construction d'une matrice requiert la connaissance de la méthode de stockage à adopter.

Les services rendus par cette classe correspondent essentiellement aux opérations effectuées sur les matrices lors des inversions de systèmes linéaires. La fonction élémentaire est donc celle du produit matrice-vecteur.

L'association entre les classes `Matrice` et `Stockage` est de type agrégation simple, le rôle des instances de la classe `Stockage` étant analogue, au niveau algébrique, à celui des instances de la classe `Discrétisation` au niveau discret.

Les classes génériques du paquetage sont donc désormais fixées. Il faut maintenant s'attacher à évaluer l'impact des différents type d'opérateurs sur leurs modèles informatiques. En effet, un opérateur linéaire ne pourra pas être traité de la même façon qu'un opérateur non linéaire. Certains traitements sont spécifiques à l'une ou à l'autre de ces catégories d'opérateurs.

La spécialisation des classes `Opérateur Continu` et `Opérateur Discret` en fonction de la linéarité de l'opérateur apparaît donc de façon naturelle.

Toutes ces classes sont des classes abstraites. Elles n'ont de réalité que par le biais des spécialisations qui en seront faites en fonction des problèmes mathématiques à traiter.

La modélisation UML de la conception détaillée du paquetage `Opérateur` est représentée par le diagramme de classe 2.6. Elle comprend les nouvelles hiérarchies spécialisant les opérateurs en opérateur linéaire et non linéaire. Comme précédemment, les noms ont été normalisé : `Operateur` correspond à `Opérateur Continu` et `Operateur_h` à `Opérateur Discret`.

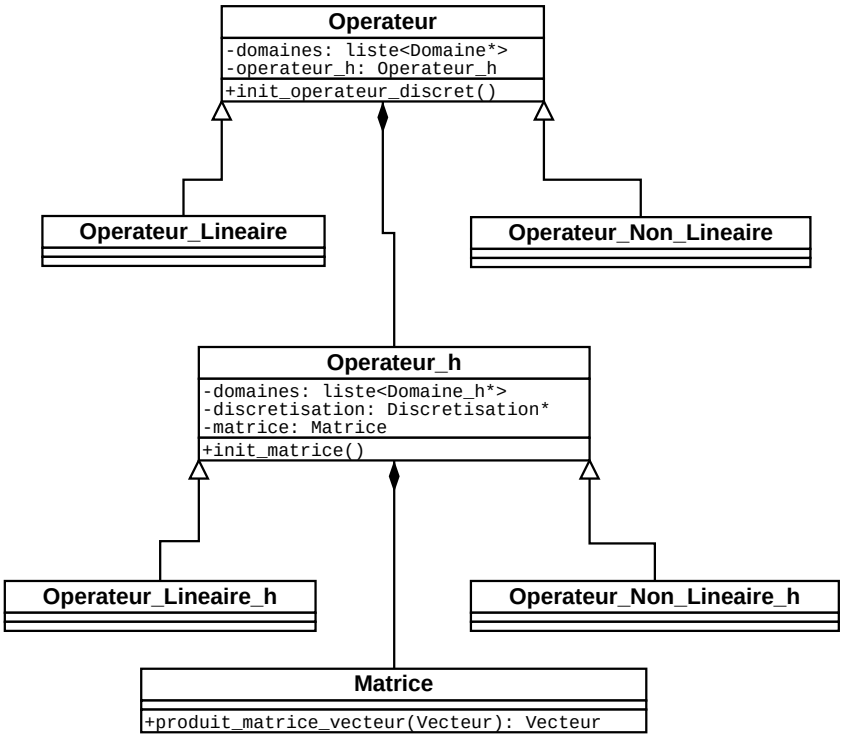

FIG. 2.6 – Diagramme de classes du paquetage `Opérateur`.

2.4.3 Conception détaillée du paquetage `Variable`

Le paquetage `Variable` comprend les classes `Variable Continue`, `Variable Discrète` et `Vecteur`. Ce paquetage est, en plus simple, construit sur le modèle du paquetage `Opérateur`. Une variable continue est définie, comme l'opérateur continu, sur un ensemble de domaines continus. Les services rendus par la classe `Variable Continue` se résument à des services de discrétisation de la variable, qui vont consister en la création de l'objet `Variable Discrète` associé.

L'association entre les classes `Variable Continue` et `Variable Discrète` est de type composition, tandis que celle existant entre `Variable Continue` et `Domaine Continu` est de type agrégation simple.

La définition d'une variable discrète nécessite la connaissance des domaines discrets approchant les domaines continus sur lesquels est définie la variable continue correspondante. La création d'une instance de `Variable Discrète` est également basé sur le choix de la méthode de discrétisation. Les services proposés par la classe `Variable Discrète` sont des services de linéarisation de la variable discrète conduisant à la création de l'objet `Vecteur` associé.

Comme pour les opérateurs, ces classes sont abstraites. Leurs réalisations concrètes se feront par des spécialisations dépendantes du problème mathématique à traiter.

La modélisation UML de la conception détaillée du paquetage `Variable` est exposée dans le diagramme de classe 2.7. La normalisation des noms fait concorder `Variable Continue` avec `Variable`, et `Variable Discrète` avec `Variable_h`.

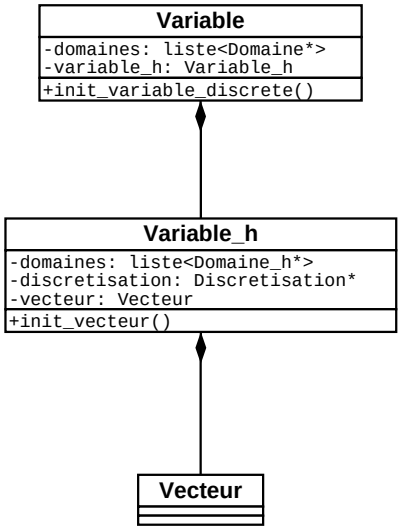

FIG. 2.7 – Diagramme de classes du paquetage `Variable`.

2.4.4 Conception détaillée du paquetage `Domaine`

Le paquetage `Domaine` contient les classes `Domaine Continu` et `Domaine Discret`.

Au niveau continu, la classe `Domaine Continu` permet la description d'un domaine de $\mathbb{R}^d$. Les services rendus par cette classe se limitent à la création de la primitive inférieure, c'est-à-dire du domaine discret associé. L'association entre ces classes est équivalente à ce qui a été déjà décrit.

Le domaine discret est la primitive la plus basse du paquetage. Il n'est donc pas responsable de la création d'une instance du niveau algébrique. Il peut représenter aussi bien un maillage de type éléments finis, qu'une grille pour les différences finies. Le point commun de ces spécialisations est qu'ils sont constitués de domaines de contrôle (mailles ...). Les services proposés par la classe générique `Domaine Discret` doivent donc permettre de parcourir ces domaines de contrôle, d'y accéder ou de les manipuler.

Comme précédemment, ces classes sont des classes abstraites dont la réalisation dépend des

spécialisations qui seront implémentées en fonction du problème mathématique intéressant le développeur.

Le diagramme de classes UML 2.8 présente la modélisation du paquetage `Domaine` en renommant `Domaine Continu` en `Domaine` et `Domaine Discret` en `Domaine_h`

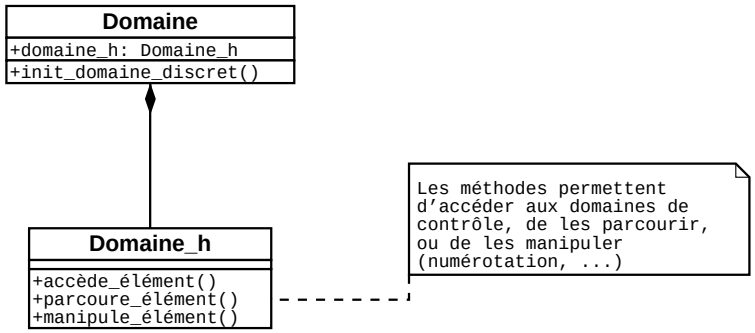

FIG. 2.8 – Diagramme de classes du paquetage `Domaine`.

2.4.5 Conception détaillée de la hiérarchie `Discrétisation`

Les instances de cette classe constitue les choix du développeur en matière de discrétisation du problème continu. Il sera donc naturel de la spécialiser en fonction des grandes familles de discrétisation : différences finies, volumes finis, éléments finis.

Les instances de la classe `Discrétisation` doivent avant tout connaître le maillage (domaine discret) sur lequel vont être calculés les opérateurs et variables discrets.

Les services proposés par la classe `Discrétisation` sont de deux types :

- des fonctions de parcours du maillage (domaines de contrôle, degrés de liberté),
- des fonctions de construction des opérateurs élémentaires restreints à un domaine de contrôle. Il s'agit de projeter localement un opérateur sur un domaine de contrôle. Cette contribution locale sera ensuite assemblé au sein de l'opérateur discret pour construire la matrice.

L'intérêt de définir une classe générique pour la discrétisation réside dans la volonté d'assurer une indépendance maximale des paquetages. Il est donc essentiel que ces spécialisations soient transparentes pour les paquetages du niveau discret.

Le polymorphisme permet alors de manipuler l'objet réel par l'intermédiaire d'une référence générique de type `Discrétisation`.

Prenons l'exemple du calcul du laplacien. Quand l'opérateur discret veut créer son équivalent algébrique, il va créer une matrice. Cette matrice va être assemblée en construisant des sous-matrices , élément du maillage par élément du maillage. Ainsi, pour les éléments finis, la fonction de construction du laplacien élémentaire, appliquées à tous les éléments (domaines de contrôle) du maillage, retournera la matrice :

$$A_k = -(\nabla\varphi_i, \nabla\varphi_j)_{(i,j)\in\mathcal{V}_k},$$

pour tous les élément k du maillage et où l'on note $\mathcal{V}_k$ l'ensemble des indices des degrés de liberté dépendant de k.

En ce qui concerne les différence finies, la matrice est simple car indépendante de la maille. Sur les points intérieurs, elle s'écrit :

$$A = \frac{1}{h^2}(1, 1, -4, 1, 1),$$

Le diagramme de classe UML 2.9 montre la modélisation de la hiérarchie de classe `Discrétisation`.

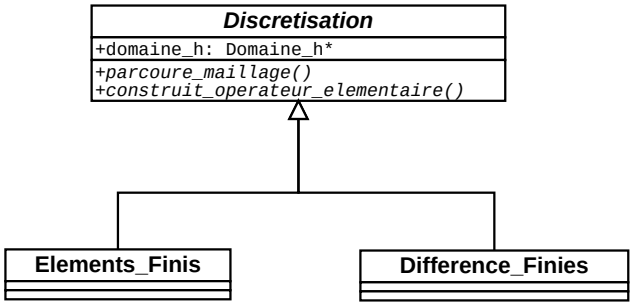

FIG. 2.9 – Diagramme de classes de la hiérarchie `Discrétisation`.

2.4.6 Conception détaillée de la hiérarchie `Stockage`

Les instances de cette classe correspondent aux choix du développeur en matière de stockage des matrices. Elle est essentielle pour optimiser la place mémoire et l'accès aux éléments.

Comme pour la classe `Discrétisation`, elle sera spécialisée en fonction des besoins et des types de matrices traitées.

Les services que doivent rendre les instances de cette classes sont :

- des fonctions d'accès aux éléments,
- des fonctions de remplissage,
- des fonctions de manipulation, essentiellement le calcul du produit matrice-vecteur.

De façon analogue au cas de la hiérarchie `Discrétisation`, il est important d'abstraire au maximum la classe `Stockage` pour assurer une meilleure modularité du niveau algébrique.

La modélisation UML de la hiérarchie de classe `Stockage` est présenté dans le diagramme 2.10

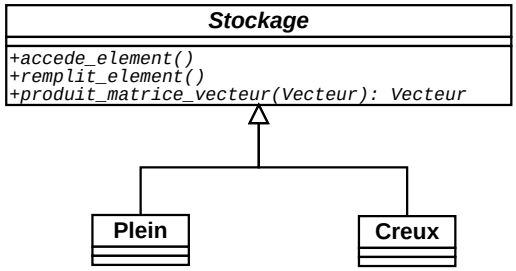

FIG. 2.10 – Diagramme de classes de la hiérarchie `Stockage`.

2.4.7 Conception détaillée des classes utilitaires

Toutes les classes mathématiques issues de la conception détaillée travaillent dans un environnement plus global permettant la gestion des flux d'entrée et de sorties.

La modélisation complète de cet environnement fait l'objet de la section 3.

Cet environnement est constitué de classes utilitaires qui permettent d'instrumenter les classes mathématiques en terme de suivi informatique (profiling, sortie standard), et de suivi du calcul (entrée et sortie de flux choisis par le développeur). Une programmation distribuée permet de rationaliser l'utilisation des moyens informatiques.

Le diagramme de composants 2.11 illustre les communications échangées entre les classes mathématiques et les classes utilitaires à travers l'architecture distribuée.

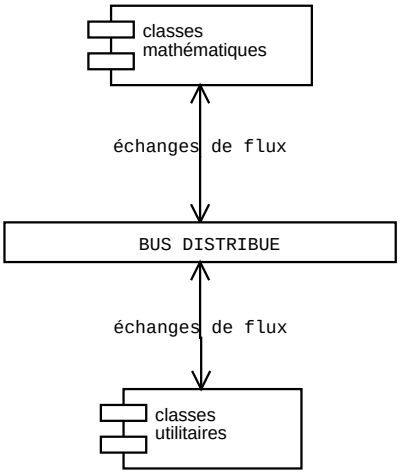

FIG. 2.11 – Diagramme de composants illustrant les communications entres les classes mathématiques et les classes utilitaires.

Chapitre 3

Analyse et conception : environnement

Introduction

La conception générale du projet s'appuie sur une approche objet de la problématique globale comprenant la partie calcul scientifique mais aussi l'ensemble des outils d'instrumentation et de développements associés. Elle nécessite la mise en oeuvre d'une méthodologie adaptée à cet objectif.

Toute méthodologie peut être considérée comme la complétion d'un formalisme et d'un processus. Dans le cadre de ce projet, notre méthodologie, basée sur le formalisme UML (Unified Modeling Language), est inspirée du processus UP (Unified Process) [**?**].

UML est un support graphique et formel pour la modélisation d'un logiciel (notations standardisées avec une sémantique précise).

UP définit un ensemble d'étapes permettant de mener à bien l'élaboration d'un logiciel :

1. conceptualisation des besoins,
2. analyse et réalisation des fonctionnalités,
3. prise en compte de l'architecture et des contraintes techniques,
4. conception,
5. implémentation et tests.

Ces différentes phases représentent une itération du cycle de vie du logiciel. Le cycle de vie itératif repose sur l'idée qu'il vaut mieux réaliser un système complexe en plusieurs fois, par évolution. La première itération permet d'arriver rapidement à un prototype du logiciel. Les itérations suivantes permettent de corriger les problèmes apparus lors de la première itération, et de modifier ou d'ajouter des fonctionnalités en fonction du retour d'expérience du prototype.

L'approche objet favorise le développement de programmes selon une démarche itérative (encapsulation, modularité ...).

3.1 Expression du besoin

Les membres de l'équipe Analyse Numérique et EDP utilisent la librairie de classes du projet GTOOCS pour développer des codes de calculs dans le cadre de leurs projets de recherche mais aussi dans le cadre de leur enseignement.

Ces activités nécessitent d'avoir un retour interactif et complet des calculs qui peuvent être important en terme de temps d'exécution. Il est donc impératif d'avoir un accès à ces données sans contrainte, depuis le poste de l'utilisateur, le code de calcul s'exécutant sur une machine dédiée de type supercalculateur. Il faut donc avoir la possibilité d'instrumenter les codes développés sur la base de la bibliothèque GTOOCS.

Les informations dont doit pouvoir disposer l'utilisateur sont les suivantes :
- sortie standard du code,
- flow trace,
- données choisies par l'utilisateur lors de l'implémentation de son code.

La visualisation de ces informations doit se faire, selon le choix de l'utilisateur et le type des données, sous forme textuelle ou sous forme graphique, éventuellement par le biais d'un logiciel de visualisation extérieur au logiciel.

Outre les fonctionnalités de suivis de calculs, l'utilisateur doit pouvoir contrôler l'exécution de son code, toujours depuis son poste de travail, par des actions de suspension, de reprise et d'arrêt du calcul.

Enfin, l'implémentation de codes de recherche rend particulièrement importante la connaissance de la documentation des programmes. Il faut donc mettre en place une structure des sources commune permettant l'extraction automatique de la documentation associée.

Toutes les données et les actions auxquelles peut accéder l'utilisateur doivent naturellement être protégées. Un utilisateur ne doit pas pourvoir accéder aux informations d'une exécution qui ne lui appartient pas. Il est donc nécessaire de recenser les utilisateurs, qui se connecteront au système via un login et un mot de passe, et d'associer chaque exécution à un utilisateur précis.

3.2 Etude préliminaire

3.2.1 Environnement de travail

On considère le système à développer comme une boîte noire intégrée à l'environnement de travail global. Le but de cet étape est de situer ce nouveau logiciel par rapport aux autres logiciels existants, ainsi que les liens pouvant exister entre eux. Cette étape permet de situer le contexte de développement, au sein de notre laboratoire, du système à réaliser. Elle permet aussi de d'appréhender les acteurs de types système qui sont en relation avec lui (détaillé en 3.2.2). Le diagramme suivant 3.1 illustre l'environnement de travail.

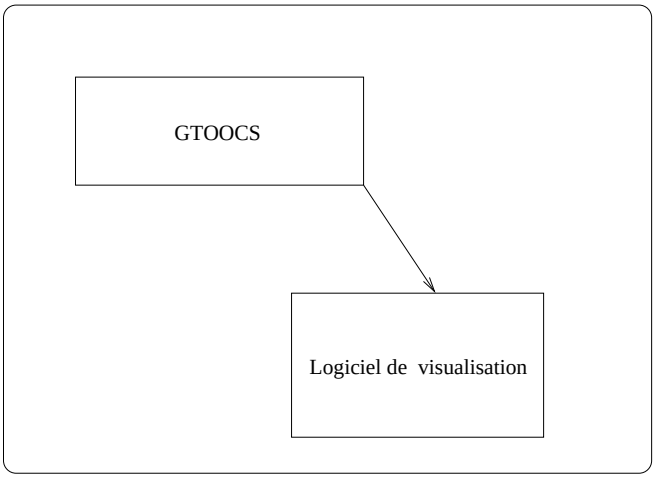

FIG. 3.1 – Environnement de travail global.

Le logiciel GTOOCS doit avoir la possibilité d'interagir avec des logiciels de visualisation qui pourront être proposés pour exploiter les résultats issus de l'instrumentation des codes.

3.2.2 Acteurs

Les acteurs regroupent toutes les entités externes au système à développer, qui sont en relation avec celui-ci. Ces acteurs peuvent être des logiciels informatiques (systèmes externes, SE) comme

des personnes physiques (Rôle Utilisateur, RU). Ils peuvent utiliser le logiciel à développer (on parle alors d'acteurs primaires) ou être utilisés par le logiciel (ils sont dits alors acteurs secondaires).

Nom	SE/RU	P/S	Définition
Utilisateur	RU	P	Cet acteur modélise une personne ayant l'autorisation d'accéder aux exécutions des codes instrumentés pour des actions de contrôle et d'exploitation des données générés par ces codes de calcul scientifique instrumentés.
Logiciel de visualisation	SE	S	Cet acteur modélise un outil extérieur au système à développer pour la visualisation graphique de données.

3.2.3 Interactions

Chacun des acteurs définies en 3.2.2 interagit avec le logiciel :
- en lui demandant un ou des services, dans ce cas, l'acteur est dit primaire,
- en lui rendant un ou des services, dans ce cas, l'acteur est dit secondaire.

Il faut maintenant définir les différentes interactions que les acteurs ont avec le système à développer.

Nom	Acteur	Définition
Suivre l'exécution	Utilisateur	L'utilisateur veut visualiser les données issues de l'exécution d'un code instrumenté.
Contrôler l'exécution	Utilisateur	L'utilisateur veut contrôler l'exécution d'un code instrumenté.
Visualiser la doc	Utilisateur	L'utilisateur veut voir la doc d'un code instrumenté.
Visualiser les données	Logiciel de visualisation	Le Logiciel utilise le Logiciel de Visualisation pour afficher les données.

3.2.4 Diagramme de contexte

Le diagramme de contexte (3.2) permet de synthétiser les informations définies dans les sections précédentes. Il est basé sur une représentation graphique utilisant les objets. Le logiciel à développer est considérer comme un objet.

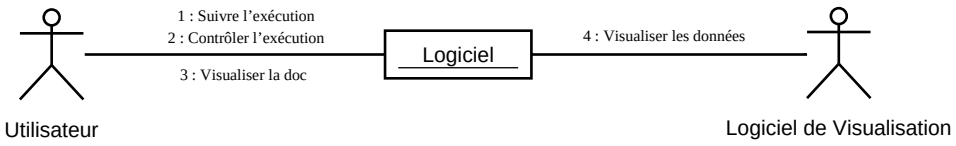

FIG. 3.2 – Diagramme de contexte.

3.3 Conceptualisation

La spécification des besoins consiste à définir les services que doit fournir le logiciel. Chacune des interactions apparaissant dans le diagramme de contexte peut correspondre à un Cas d'Utilisation (Use Case). Un UC est une fonctionnalité du logiciel nécessitant un travail collaboratif entre le système et les acteurs. Chaque Cas d'Utilisation donne lieu à une description textuelle qui comprend entre autres le scénario dit nominal. Celui-ci déroule l'ensemble des actions à réaliser pour atteindre l'objectif de l'UC.

A côté de ce scénario principal, qui décrit le bon fonctionnement du logiciel, peuvent intervenir des scénarii secondaires servant à traiter les cas d'erreurs (dysfonctionnements).

Les UC principaux qui réalisent toutes les fonctionnalités du système sont appelés UC de niveau tâche. Les étapes de leur scénario principal peuvent constituer elles-mêmes des UC, appelés UC de niveau fonction. Les UC de niveau tâche correspondent en général aux interactions relatives aux acteurs primaires.

Les différents Cas d'Utilisation du système sont décrits dans les sections suivantes. On ne considère ici que la première itération du processus de conception, c'est-à-dire qu'on ne traite que les scénarii principaux de chaque UC. Les scénarii secondaires sont établis lors des itérations suivantes.

3.3.1 UC niveau tâche

UC1 : Suivre l'exécution

Nom : Suivre l'exécution.

But : L'utilisateur veut visualiser les données issues de l'exécution d'un code instrumenté.

Acteur primaire : Utilisateur.

Acteur secondaire : Logiciel de visualisation.

Scénario principal :

1. Le système identifie l'utilisateur. [UC4]
2. L'utilisateur accède à l'exécution du code instrumenté qui l'intéresse. [UC5]
3. L'utilisateur visualise les données issues de cette exécution d'un code instrumenté, éventuellement grâce au logiciel de visualisation. [UC6]

UC2 : contrôler l'exécution

Nom : Contrôler l'exécution.

But : L'utilisateur veut contrôler l'exécution d'un code instrumenté.

Acteur primaire : Utilisateur.

Scénario principal :

1. Le système identifie l'utilisateur. [UC4]
2. L'utilisateur accède à l'exécution du code instrumenté qui l'intéresse. [UC5]
3. L'utilisateur choisit l'action à effectuer (suspendre/arrêter/redémarrer).
4. Le logiciel applique l'action choisie à l'exécution.

UC3 : Visualiser la doc

Nom : Visualiser la doc.

But : L'utilisateur veut voir la doc d'un code instrumenté.

Acteur primaire : Utilisateur.

Scénario principal :

1. Le système identifie l'utilisateur. [UC4]
2. L'utilisateur accède à l'exécution qui l'intéresse. [UC5]
3. L'utilisateur choisit un des fichiers sources de cette exécution pour lequel il veut la doc.
4. Le logiciel affiche la documentation relative à ce source.

3.3.2 UC niveau fonction

Les Cas d'Utilisation de niveau fonction permettent de détailler certains pas des scénarii nominaux des Cas d'Utilisation de niveau tâche.

UC4 : Identifier l'utilisateur

Nom : Identifier l'utilisateur

But : Le logiciel doit identifier l'utilisateur qui veut se connecter.

Niveau : Fonction

Acteur primaire : Utilisateur

Scénario principal :

1. L'utilisateur fournit les informations d'accès au logiciel.
2. Le logiciel identifie l'utilisateur.

UC5 : Accéder à une exécution

Nom : Accéder à une exécution.

But : L'utilisateur accède à l'exécution qui l'intéresse.

Niveau : Fonction

Acteur primaire : Utilisateur

Pré-condition : Utilisateur identifié par le logiciel. [UC4]

Scénario principal :

1. Le logiciel affiche la liste des exécutions en cours.
2. L'utilisateur choisit l'exécution qui l'intéresse.
3. Le logiciel vérifie l'autorisation d'accès à cette exécution.

UC6 : Visualiser les données d'une exécution

Nom : Visualiser les données d'une exécution.

But : L'utilisateur accède aux données d'une exécution et les visualise.

Niveau : Fonction

Acteur Primaire : Utilisateur

Acteur Secondaire : Logiciel de visualisation

Pré-condition : L'utilisateur a accédé à l'exécution. [UC5]

Scénario principal :

1. Le logiciel affiche la liste des données relatives à l'exécution.
2. L'utilisateur choisit la donnée à visualiser.
3. L'utilisateur choisit le mode de visualisation pour cette donnée (texte, graphique, statistiques).
4. Le logiciel affiche la donnée, éventuellement grâce au logiciel de visualisation.

3.3.3 Diagramme de cas d'utilisation

Les différents cas d'utilisation définis en 3.3.1 et en 3.3.2 peuvent être représentés graphiquement par un diagramme UML qui permet de visualiser les UC prédominants (en général, les UC les plus inclus) (3.3).

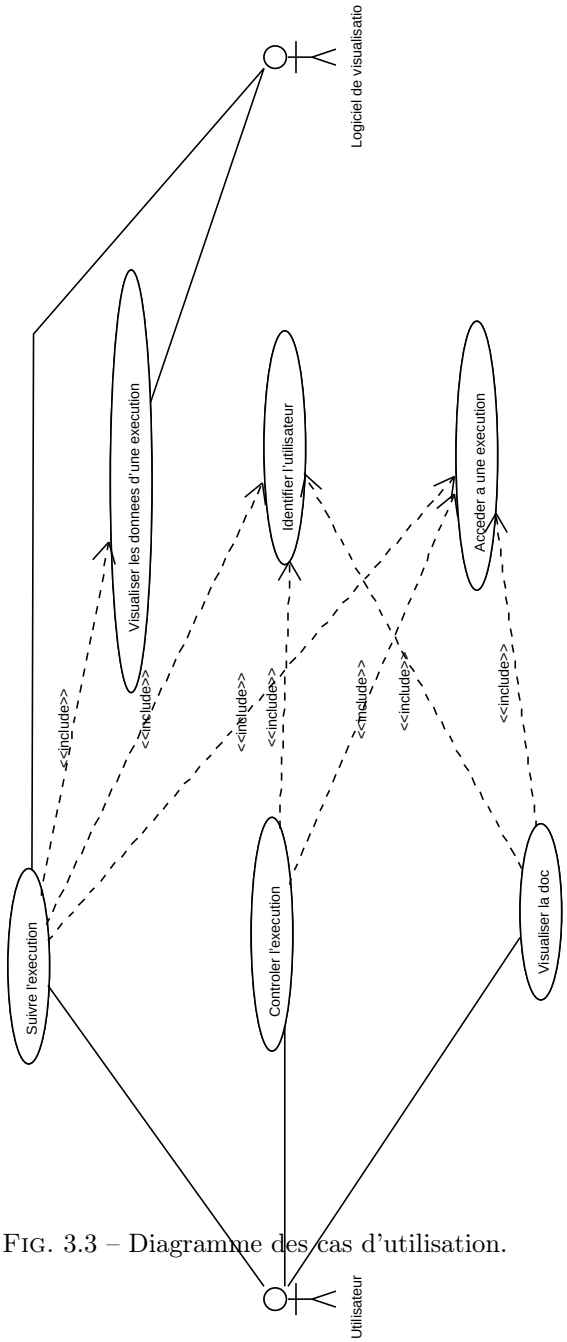

FIG. 3.3 – Diagramme des cas d'utilisation.

3.3.4 Spécification du besoin relatif à l'interface graphique

La partie instrumentation du logiciel GTOOCS a pour vocation d'interagir interactivement avec l'utilisateur. Cela suppose nécessairement la mise en place d'une interface graphique qui permettra d'accéder à toutes les fonctionnalités du logiciel. Il est donc indispensable de spécifier les besoins en matières d'interface graphique, au même titre que ce qui a été fait précédemment pour les fonctionnalités propres du système.

Les objets graphiques se distinguent des autres objets du logiciel par la présence du suffixe *"View"* dans le nom de leur classe. Ces contextes d'interactions sont présentés sous forme de classe, dans un diagramme UML 3.4.

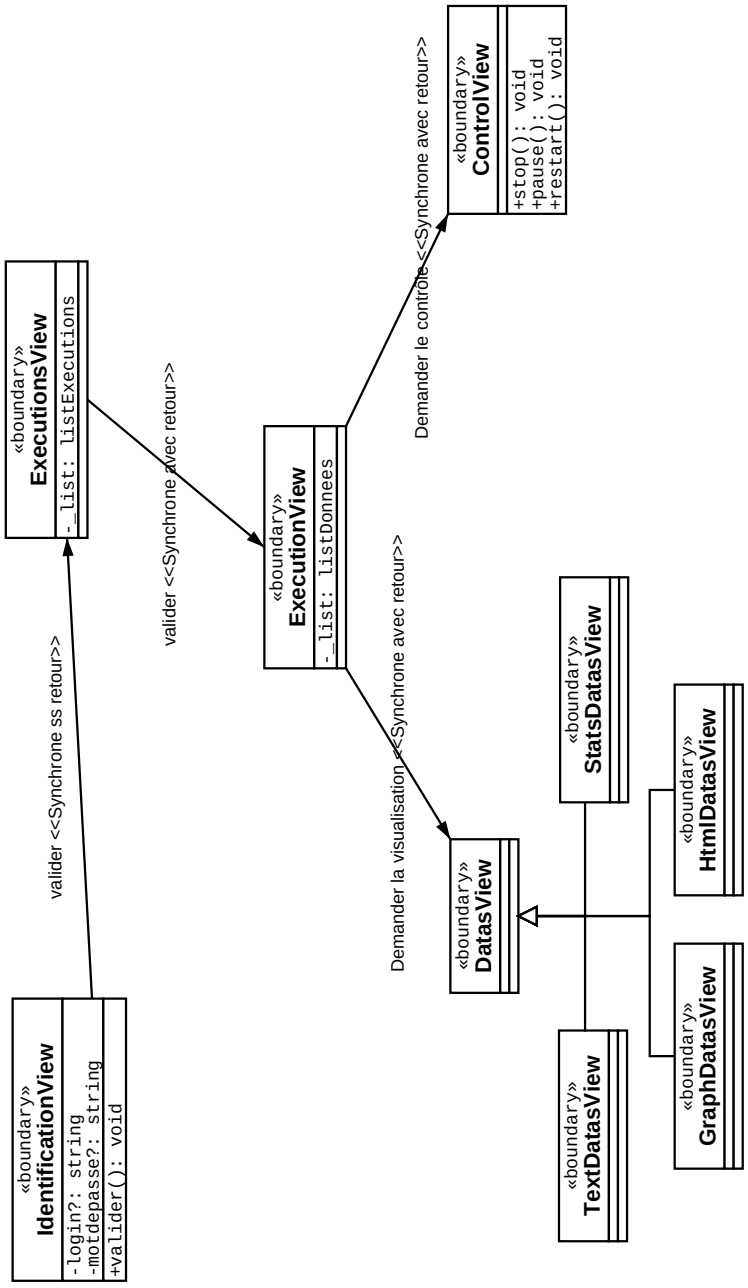

FIG. 3.4 – Diagramme U.M.L. du besoin IHM.

3.4 Analyse

L'objectif de l'analyse est de comprendre et de structurer le logiciel. La phase d'analyse consiste donc à modéliser le problème de façon orienté objet. Il faut recenser de précisemment :

- la liste des principales entités du système (classes et instances),
- les relations de toutes sortes (héritage, composition, communication, ...) qui existent entre ces différentes entités,
- les aspects dynamiques (envois de messages, ...),
- les aspects fonctionnels (flux, traitements, ...).

On peut distinguer deux phases :

- l'analyse du domaine qui consiste à identifier les objets du problème,
- l'analyse applicative qui revient à identifier les objets de contrôles liés au logiciel, et qui seront responsables de la réalisation des UC.

Ces deux étapes sont regroupés lors de l'élaboration du modèle d'analyse, qui en s'appuyant sur un découpage métier des entités, permet de combiner toutes les informations.

Cette ultime phase de l'analyse permet d'obtenir une vue quasi-idéale interne du logiciel, c'est-à-dire sans trop tenir compte des contingences de conception.

3.4.1 Analyse du domaine

La phase d'analyse commence par recenser et détailler l'ensemble des substantifs apparaissant dans la phase de conception. Il s'agit de représenter tout ce qui est manipulé par le logiciel.

Une première représentation textuelle consiste en l'élaboration d'un glossaire permettant la définition des entités constituant le logiciel.

Nom	**Définition**
Exécution Execution	Process correspondant à l'exécution d'un code de calcul instrumenté.
Utilisateur EndUser	Personne possèdant les autorisations d'accès pour accèder aux exécutions des codes instrumentés.
CodeInstrumenté InstrumentedCode	Code de calcul intégrant les mécanismes permettant l'exploitation distante des données.
LogicielVisualisation VisualTool	Outil permettant de visualiser graphiquement des données.
ActionContrôle ControlAction	Action permettant à l'utilisateur d'agir sur l'exécution d'un code instrumenté.
Suspendre Pause	Action de suspendre l'exécution d'un code.
Arrêter Stop	Action d'arrêter l'exécution d'un code.
Redémarrer Restart	Action de redémarrer l'exécution d'un code.
Source Source	Fichier source faisant partie des sources d'un code instrumenté.
Doc Doc	Fichier de documentation relatif à un Source d'un code instrumenté.
InformationsAccès AccessData	Informations nécessaires à un Utilisateur pour accéder à la liste des exécutions des codes instrumentés.
ListeExecutions ExecutionList	Liste de toutes les exécutions des codes intrumentés disponibles à un instant donné.
AutorisationAccès AccessAuthorization	Autorisation propre à chaque exécution pour accéder à ses données dans un souci de confidentialité.
ListeDonnées	Liste de toutes les Données disponibles relatives

DataList	à une exécution d'un code instrumenté.
Donnée Data	Information disponible relative à une exécution d'un code instrumenté.
ModeVisualisation VisualizationMethod	Technique de visualisation choisi pour visionner une Donnée.

L'établissement de ce glossaire est complété par un diagramme de classe UML (3.5). Pour des questions de généricité, les noms des classes et des objets du diagramme correspondent aux substantifs anglais du glossaire.

3.4.2 Analyse applicative

Les classes définies lors de l'analyse du domaine servent pour l'analyse applicative qui consiste à réaliser les cas d'utilisation grâce à des diagrammes de séquences. On dissocie les responsabilités de chaque objet pour faciliter la mise en oeuvre d'une architecture distribuée. On distingue donc :

- les objets frontières , notés

, qui sont de manière générale des objets d'interface (Interface graphique, Interface vers un autre logiciel ou un système externe),
- les objets contrôles, notés

, qui jouent un rôle de collaboration avec les autres objets. Ils portent en général la responsabilité de la réalisation des UC.
- les objets entités, notés

, sont les objets métiers auxquels sont associées des données.

L'objectif de l'analyse applicative est donc de définir les responsabilités de chaque classe et d'identifier les objets de contrôle pour chacun des cas d'utilisation définis en 3.3.1 et 3.3.2.

Réalisation de l'UC 4 : Identifier l'utilisateur

Le diagramme 3.6 représente le déroulement dynamique de ce cas d'utilisation.

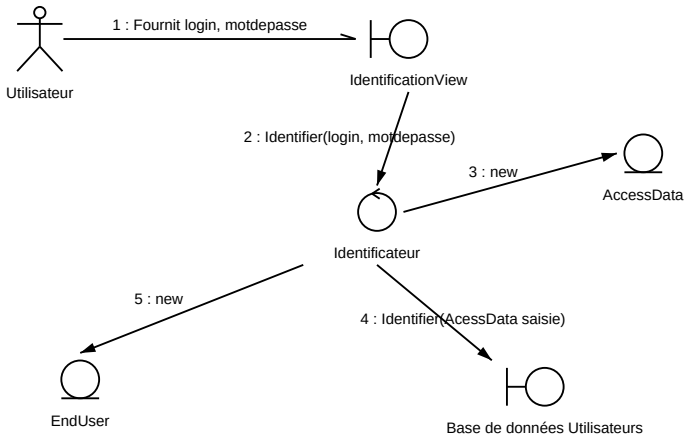

FIG. 3.6 – Diagramme de séquence du cas d'utilisation 4.

Réalisation de l'UC 5 : Accéder à une exécution

On suppose que la pré-condition de l'UC est réalisée et donc que l'utilisateur a été identifié par le logiciel. Le diagramme 3.7 illustre ce cas d'utilisation.

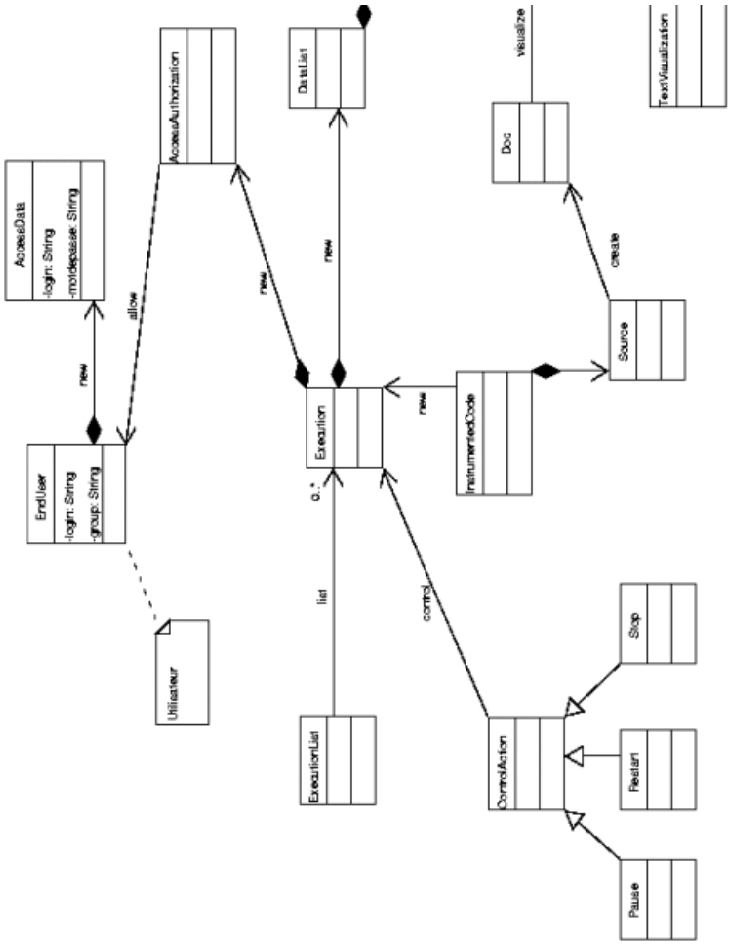

FIG. 3.5 – Diagramme des classes issues de l'analyse du domaine.

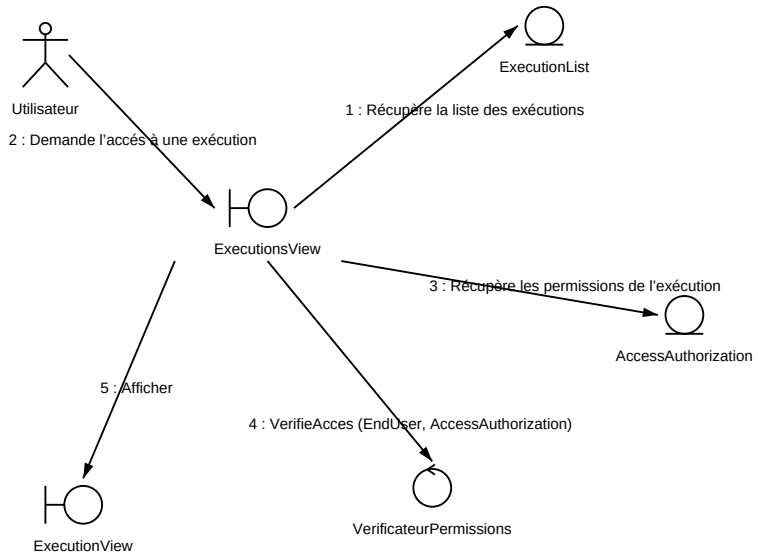

FIG. 3.7 – Diagramme de séquence du cas d'utilisation 5.

Réalisation de l'UC 6 : Visualiser les données d'une exécution

On suppose que la pré-condition de l'UC est réalisée, et donc que l'utilisateur a accédé à l'exécution qui l'intéresse. Le diagramme 3.8 présente le déroulement du scénario de cet UC.

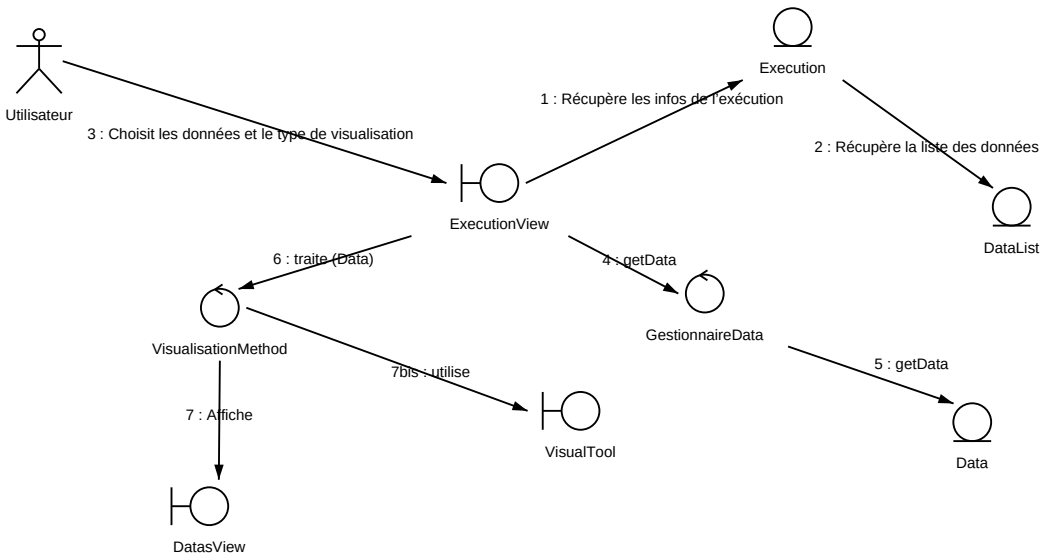

FIG. 3.8 – Diagramme de séquence du cas d'utilisation 6.

Réalisation de l'UC 1 : Suivre l'exécution

Cet UC est obtenu par la réalisation de ces UC inclus c'est-à-dire les UC 4, 5 et 6. Il n'y a donc pas de diagramme propre à ce cas d'utilisation.

Réalisation de l'UC 2 : Contrôler l'exécution

On suppose que les deux premiers pas du scénario nominal sont réalisés. Ces étapes correspondent aux cas d'utilisation 4 et 5. On considère donc que l'utilisateur a accédé à l'exécution qui l'intéresse. Ce scénario est illustré en 3.9.

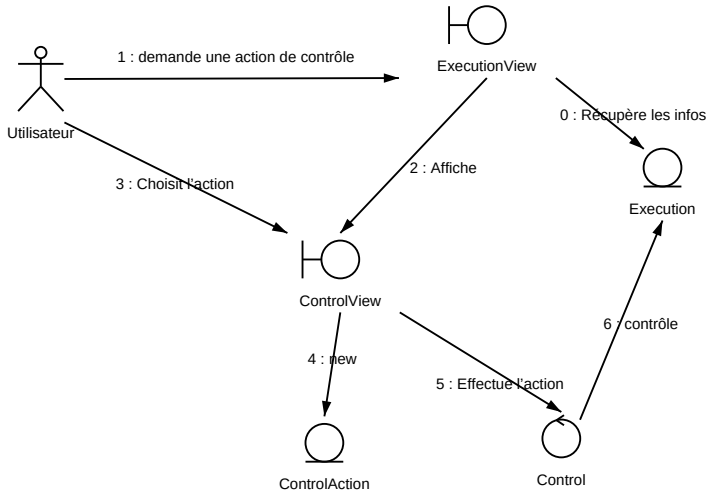

FIG. 3.9 – Diagramme de séquence du cas d'utilisation 2.

Réalisation de l'UC 3 : Visualiser la doc

Comme pour le cas précédent, on suppose que les deux premiers pas du scénario nominal sont réalisés. Ces étapes correspondent aux cas d'utilisation 4 et 5. On considère donc que l'utilisateur a accédé à l'exécution qui l'intéresse. Le diagramme de séquence est donné en 3.10.

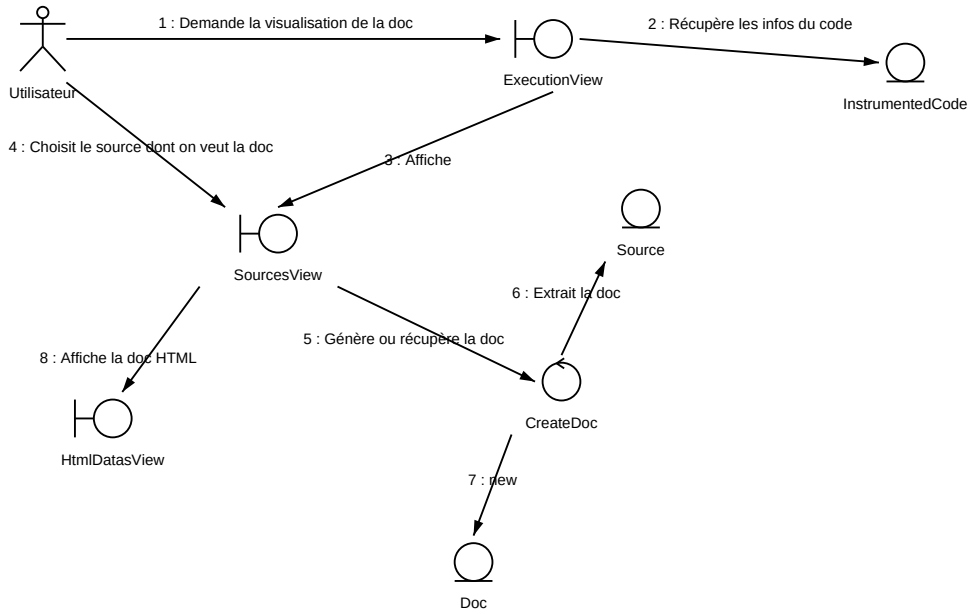

FIG. 3.10 – Diagramme de séquence du cas d'utilisation 3.

3.4.3 Modèle d'analyse

Les diagrammes élaborées lors des phases 3.4.1 et 3.4.2 déterminent une première structure du logiciel. Il s'agit maintenant d'élaborer un diagramme de classes (3.11) reprenant les entités définies précédemment, agencées selon un découpage métier. Ce découpage est réalisé en agglomérant les classes relatives aux mêmes substantifs.

Les stéréotypes de classes puis les catégories (paquetages) permettent alors d'organiser le logiciel.

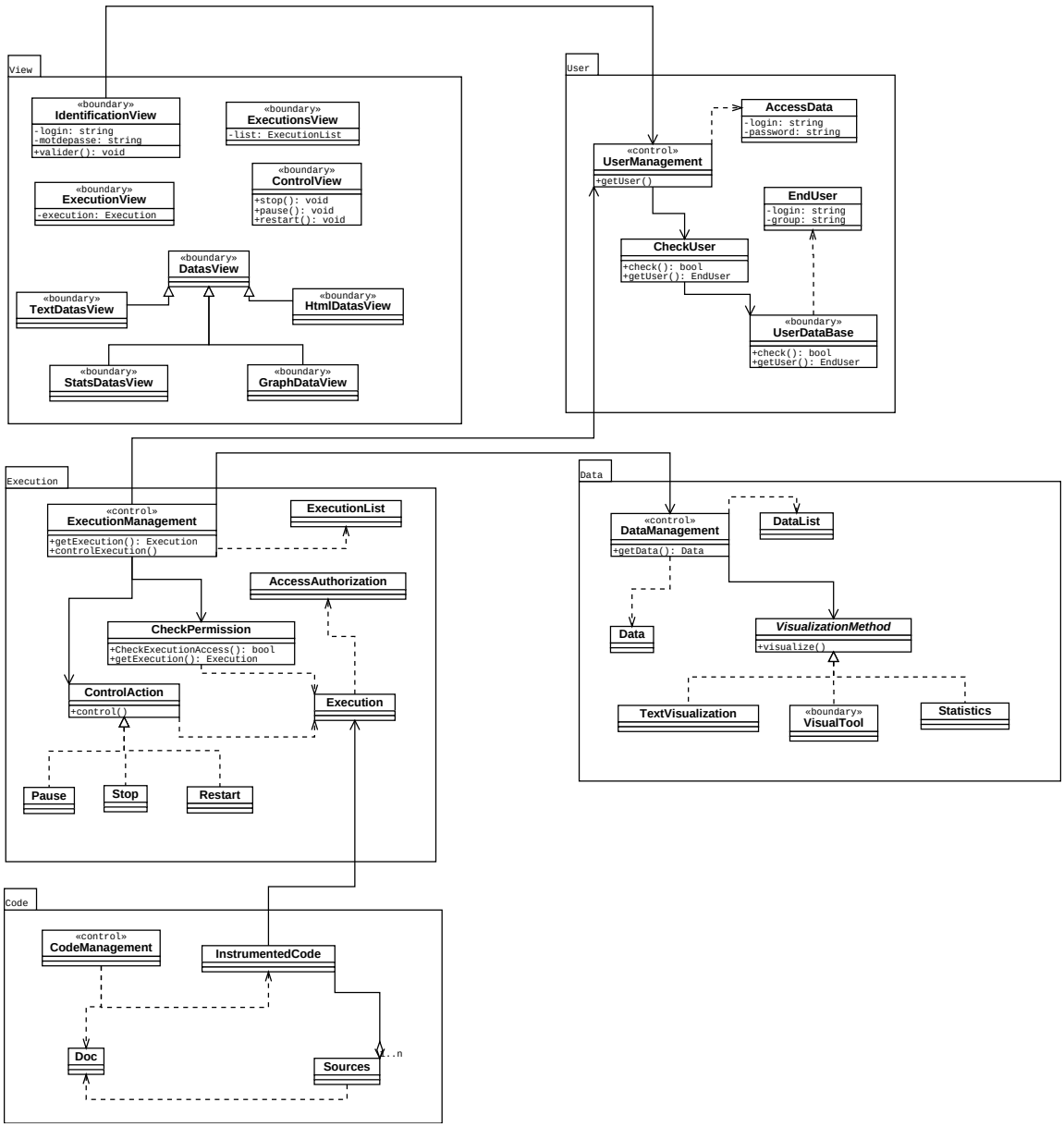

FIG. 3.11 – Diagramme de classes du modèle d'analyse.

3.5 Architecture

3.5.1 Architecture logicielle

Cette étape permet de prendre en compte l'architecture et les contraintes techniques. On peut considérer que le projet comporte trois grandes parties :

- la présentation, c'est-à-dire l'IHM,
- les traitements, c'est-à-dire le calcul lui-même,
- les données, c'est-à-dire tout ce qui doit être conservé pour le suivi et le contrôle du calcul, l'exploitation des résultats ...

Chacune d'elles nécessite des besoins techniques particuliers qu'il est important de prendre en compte :

- l'IHM doit être multiplateforme,
- le calcul demande des performances machines importantes,
- les données doivent toujours être à disposition. Elles sont distribuées aux machines d'exploitation. Celles-ci doivent donc avoir connaissance de la localisation du stockage de ces données.

A partir de ces contraintes, on peut définir une première vue logique de l'architecture du logiciel. Les objets métiers (*Code*) sont regroupés dans le paquetage *Calculation*. Les objets circulants et persistants (*Data*, *User*, *Execution*) sont contenus dans le paquetage *AllDatas*. Les composants miroirs des objets précédents sont contenus dans le paquetage *IHM*.

Le diagramme 3.12 présente ces différents paquetages du modèle d'analyse projetés sur une architecture de type 3 tiers (IHM, traitements et données).

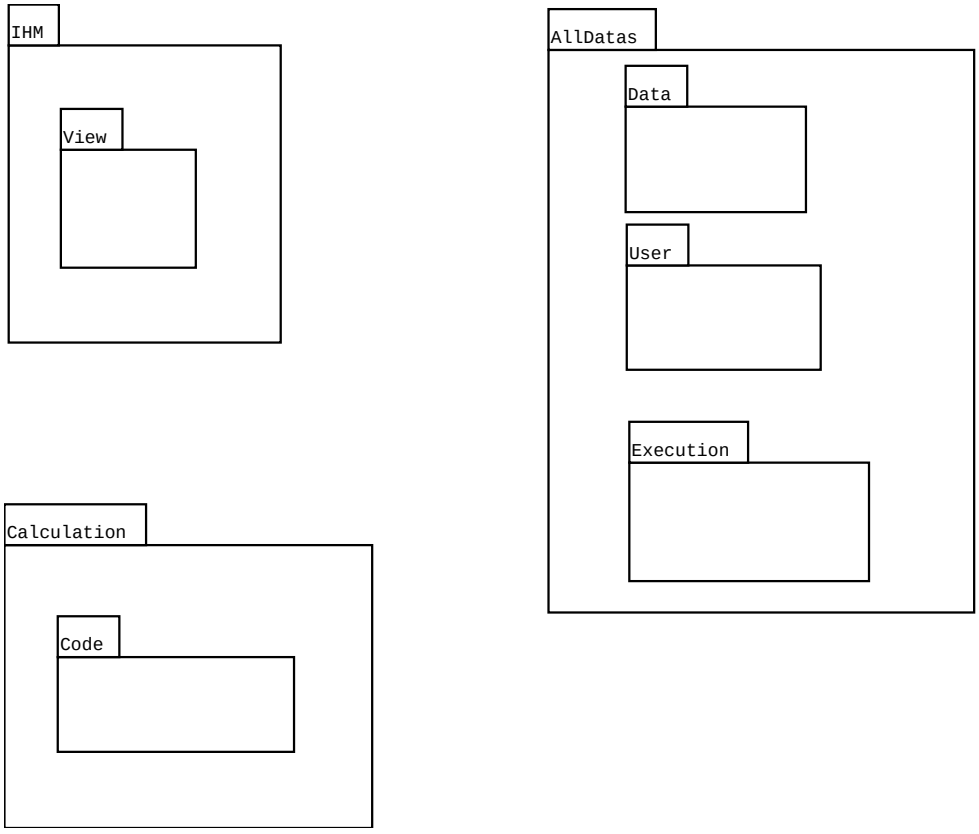

FIG. 3.12 – Modèle d'analyse projeté sur une architecture de type 3 tiers.

A partir cette décomposition 3 tiers, il est nécessaire de faire les bons choix techniques, en fonction de l'état de l'art et de l'évolution prévue du logiciel. La mise en oeuvre d'un modèle d'architecture distribuée semble nécessaire. Ce modèle doit répondre à des critères techniques précis explicités dans les sections suivantes.

3.5.2 Architecture matérielle

Choix techniques

Choix des systèmes d'exploitation Les systèmes d'exploitation sur lesquels doivent fonctionner le logiciel sont hétérogènes. En effet, le poste de l'utilisateur final peut très bien utiliser Microsoft Windows, tandis que le calcul s'effectue sur une machine Unix dédiée. Il est donc important que les choix techniques qui vont être fait dans les sections suivantes soient compatibles avec cette diversité.

Choix des protocoles réseaux Le protocole réseau qui permet de répondre à la contrainte exprimée en 3.5.2 est le protocole TCP/IP. Ainsi, le choix du modèle de distribution doit prendre en compte cette exigence supplémentaire.

Choix des langages de programmation L'interface graphique s'exécute sur la machine de l'utilisateur final. Le langage de programmation de cette interface doit donc être multi-os conformément à la contrainte exprimée en 3.5.2. Pour assurer un portage maximal, le choix du langage Java paraît adapté.

Le calcul est effectué le plus souvent sur une machine dédiée de type supercalculateur. Le langage de programmation principal doit avant tout être objet et performant. Cependant, il est parfaitement envisageable d'utiliser des langages de programmation secondaires non objet pour optimiser l'efficacité des calculs.

Le langage objet portable, performant et standardisé le plus intéressant est le C++. D'autre part, il s'interface facilement avec des langages plus axés calculs numériques comme le fortran 77 ou le fortran 90.

Quelques outils, notamment concernant la génération de la documentation issue des sources des programmes, nécessitent la mise en oeuvre d'un langage de script pour lequel le choix se porte sur Perl.

La base des utilisateurs est implémenté sous mysql, base de données gratuite et interfaçable avec le C++, comme avec Java.

Choix du modèle de distribution Le middleware d'architecture distribué qui permettra d'assurer une répartition 3 tiers des modules constituants le logiciel doit donc impérativement vérifier les critères suivants, définis par les paragraphes précedents :

- multi-systèmes d'exploitation et multi-plateformes,
- distribution sur protocole réseau TCP/IP,
- compatible avec les langages Java et C++.

On peut ajouter à ces contraintes les points suivants, propre à notre environnement de travail :

- Le modèle de distribution doit être objet, conformément à la structure globale du logiciel.
- Ce protocole doit être du domaine public, donc gratuit et open-source.
- Il doit être normalisé et reconnu, afin d'assurer une certaine pérennité du logiciel.

Ces contraintes nous ont conduit à faire le choix du middleware CORBA, qui répond effectivement aux besoins précédents :

- Il existe de nombreuses implémentations du bus CORBA du domaine public.
- Ce middleware orienté objet est avant tout une norme spécifiée par un consortium international important (l'OMG : Object Management Group) qui lui assure une certaine pérennité.
- De nombreuses implémentations du domaine public sont multi-plateformes, et la norme a été spécifiée de telle sorte que deux implémentations sachent communiquer entres elles.

- Les principaux langages d'utilisation de CORBA sont C++ et Java, qui peuvent donc autoriser une encapsulation des langages tels que le fortran 77 et le fortran 90.
- Les communications entre bus CORBA par l'intermédiaire du protocole TCP/IP sont parfaitement spécifiée et opérationnelles. Deux machines distantes peuvent donc aisément communiquer par ce moyen.

Les autres middlewares d'architecture distribuée (RMI, DCOM ...) ne répondent pas complètement aux contraintes exprimées (problème de portage, de langage ...).

Vue de l'architecture physique

A partir des choix techniques exprimés dans les sections précédentes, on peut décomposer le logiciel en différents modules :

- le module client pour l'IHM (paquetage *View*),
- le module serveur de données (paquetages *User*, *Data* et *Execution*),
- le module de calcul (paquetage *Code*).

Ces modules interagissent entres eux selon le diagramme 3.13.

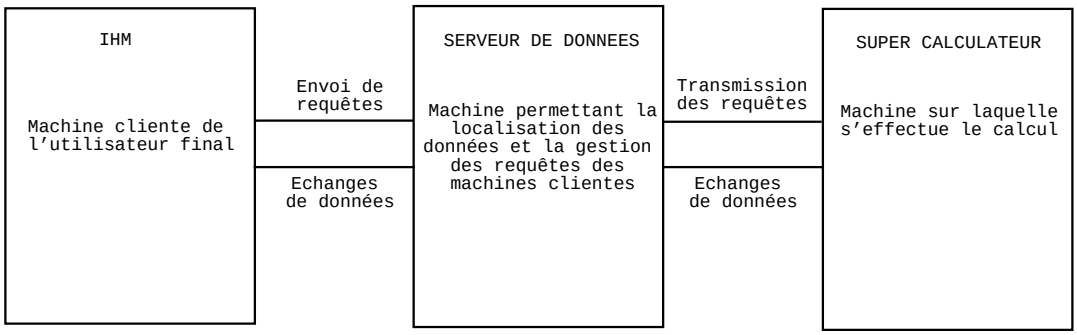
IHM
Machine cliente de l'utilisateur final
Envoi de requêtes
Echanges de données
SERVEUR DE DONNEES
Machine permettant la localisation des données et la gestion des requêtes des machines clientes
Transmission des requêtes
Echanges de données
SUPER CALCULATEUR
Machine sur laquelle s'effectue le calcul

FIG. 3.13 – Schéma de l'architecture physique.

Cette décomposition doit assurer les contraintes techniques suivantes :

- partage des données,
- persistance des données,
- sécurisation des données,
- synchronisation entre le module client et le module de calcul,
- pas de copie des données : un seul objet par donnée, car celles-ci peuvent être très importantes en terme de quantité.

Ces éléments conduisent au diagramme de déploiement 3.14 illustrant une vue de l'architecture physique.

3.6 Conception préliminaire

La conception correspond à une description de haut niveau du produit, en termes de modules et de leurs interactions. Il s'agit de concevoir les solutions techniques à mettre en oeuvre, en terme de modélisation (conception préliminaire) et de choix d'implémentation (conception détaillée).

Cette étape consiste ainsi à amalgamer les objets et classes d'analyse dans l'architecture technique.

Cette phase ne se situe plus seulement au niveau des concepts. Il faut déterminer comment manipuler ces concepts, donc comment les implémenter. Pour cela, on s'appuie sur des patrons achitecturaux (Design Pattern, [**?**]), qui sont des modèles de conception réutilisables. On utilise notamment le pattern Façade qui sert à définir une interface unique pour un sous-système.

Le déploiement du sous-système *AllDatas* regroupant l'ensemble des données communiquantes nécessite sa distribution sur plusieurs noeuds :

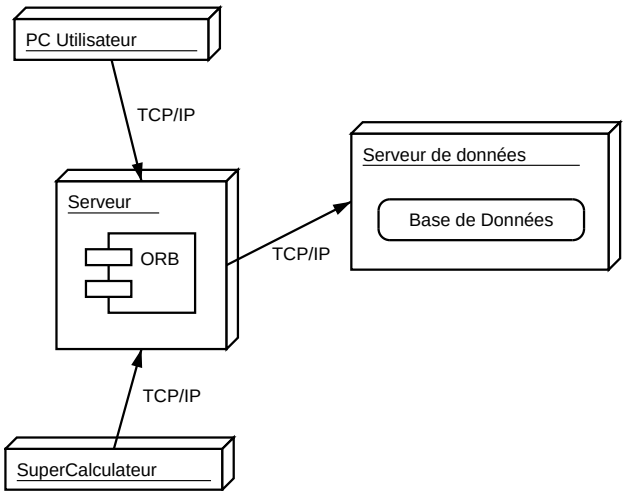

FIG. 3.14 – Diagramme de déploiement sur l'architecture.

- le noeud du serveur de données qui connaît la localisation physique des données,
- le noeud du supercalculateur qui met à jour les données par l'intermédiaire du serveur et récupère éventuellement les modifications qui ont eu lieu sur le serveur,
- le noeud de la machine de l'utilisateur final, qui récupère et éventuellement agit sur les données localisées grâce au serveur.

Ainsi, il est nécessaire de le décomposer en trois nouveaux sous-systèmes afin de prendre en compte le déploiement illustré en 3.5.2.

3.6.1 Module User

Conformément à la remarque ci-dessus, le sous-système *User* est décomposé en trois nouveaux sous-systèmes (diagramme 3.15) :

- le module *DBUser,* déployé sur le noeud serveur de données, qui gère la base de données utilisateur et auquel se réfère les deux autres modules,
- le module *Owner,* déployé sur le noeud supercalculateur, qui permet, par un référencement sur le module précédent de connaître le propriétaire du calcul,
- le module *EndUser,* déployé sur le noeud machine terminale, qui permet l'identification de l'utilisateur final, par rapport à la base de données du serveur.

3.6.2 Module Data

Comme pour le module précédent, le module *Data* est déployé sur l'architecture du système selon les trois nouveaux sous-systèmes suivants (diagramme 3.16) :

- le module *Stream* sur le noeud serveur de données, qui englobe les objets de type flux communiquants et auxquel se réfèrent les objets des deux autres sous-systèmes,
- le module *RStream* qui correspond aux flux distants (Remote Stream), déployé sur le noeud supercalculateur, dont les objets sont la base de l'instrumentation des codes,
- le module *VStream* (Visualize Stream), qui définit les classes pour la visualisation des flux sur la machine de l'utilisateur final.

3.6.3 Module Execution

Le dernier module du paquetage *AllDatas* est également déployé sur l'architecture techinque du logiciel en trois sous-modules (diagramme 3.17) :

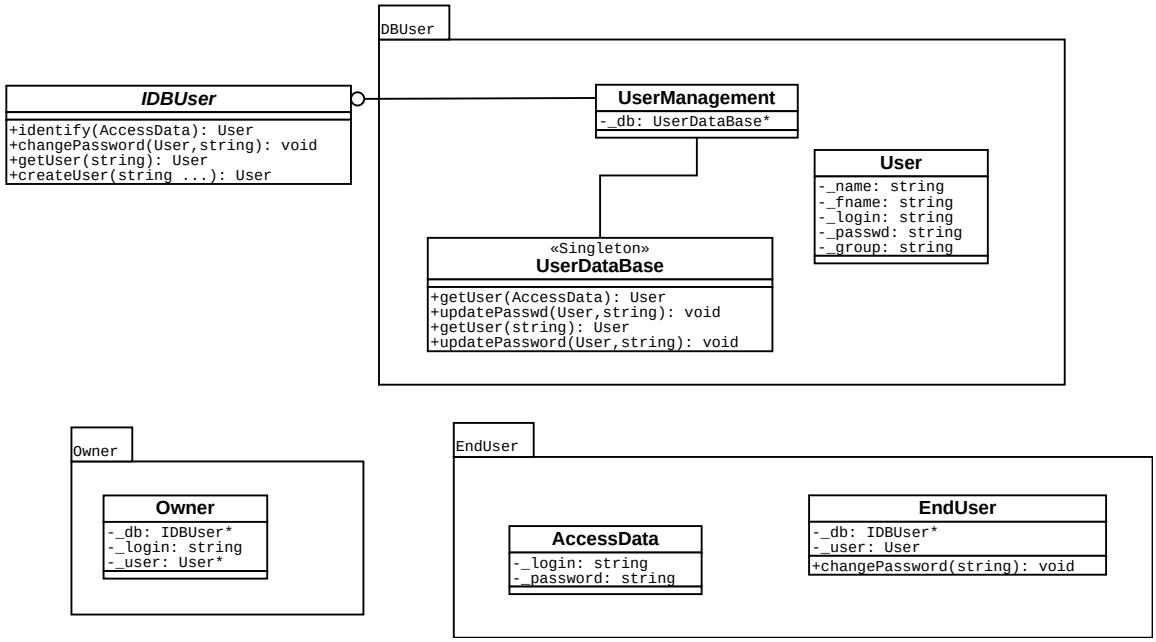

FIG. 3.15 – Diagramme U.M.L. du sous-système *User*.

- le module *Execution* qui référence les exécutions sur le serveur de données,
- le module *Process* qui correspond aux exécutions des codes instrumentés sur le supercalculateur,
- le module *Calcul* sur la machine de l'utilisateur final qui permet d'accéder aux objets du module *Execution* du serveur de données.

3.6.4 Module View

Comme la plupart des interfaces graphiques, nous allons utiliser l'architecture Model-View-Controller (MVC) pour modéliser l'IHM du logiciel. Cette architecture est composée de trois parties :

- le modèle, qui est la partie décrivant les données à afficher,
- la vue, qui est la représentation graphique de ces données,
- le contrôleur, qui est la partie qui traite des interactions du composant avec l'utilisateur.

Nous pouvons définir plusieurs types de modèles correspondant aux entités interagissant avec l'utilisateur par le biais de l'interface graphique :

- Modèle relatif aux données de l'utilisateur et aux données d'identification (*EndUser* et *AccessData*),
- Modèle relatif aux données des exécutions en cours (*YPCalculs*),
- Modèle relatif aux données d'une exécution particulière (*Calcul*),
- Modèle relatif à un flux de données particulier d'une exécution précise (*VIOstream*).

Chacun de ces modèles a été définis dans les sections précédentes via les sous-systèmes déployés sur le noeud machine de l'utilisateur final.

Selon l'architecture MVC, il faut associer à chaque modèle une vue et un contrôleur. Dans notre cas, l'objet contrôleur est pris en charge par le modèle comme l'illustre les diagrammes ci-dessous.

Le diagramme 3.18 correspond aux classes générales de vue et de gestion.

Le diagramme 3.19 correspond aux vues liées à l'objet utilisateur.

Le diagramme 3.20 illustre les vues associées aux pages jaunes des exécutions.

Le diagramme 3.21 concerne les vues relatives à une exécution particulière.

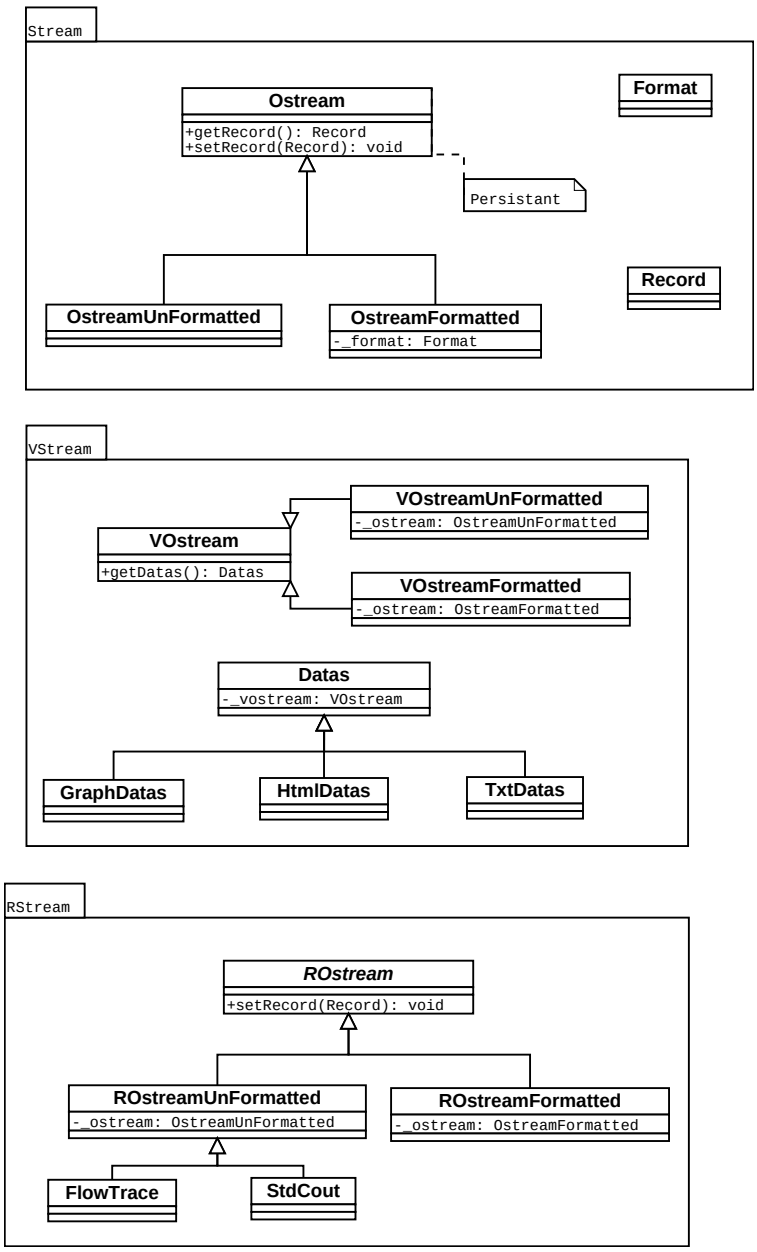

FIG. 3.16 – Diagramme U.M.L. du sous-système *Data*.

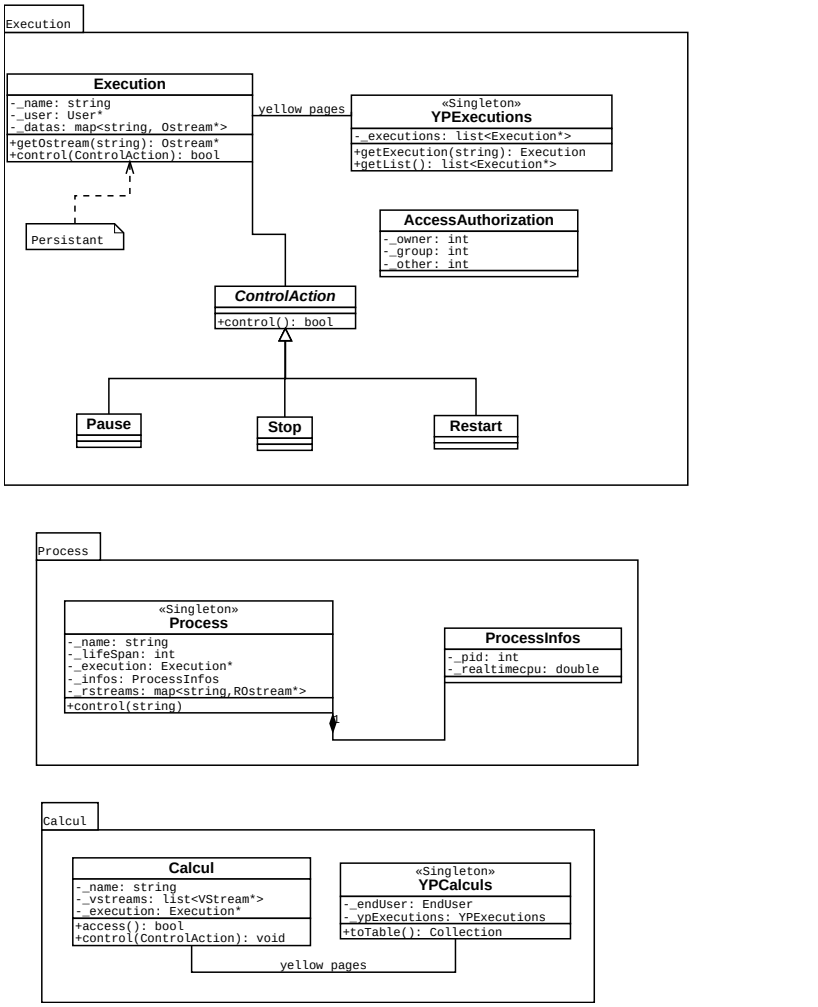

FIG. 3.17 – Diagramme U.M.L. du sous-système *Execution*.

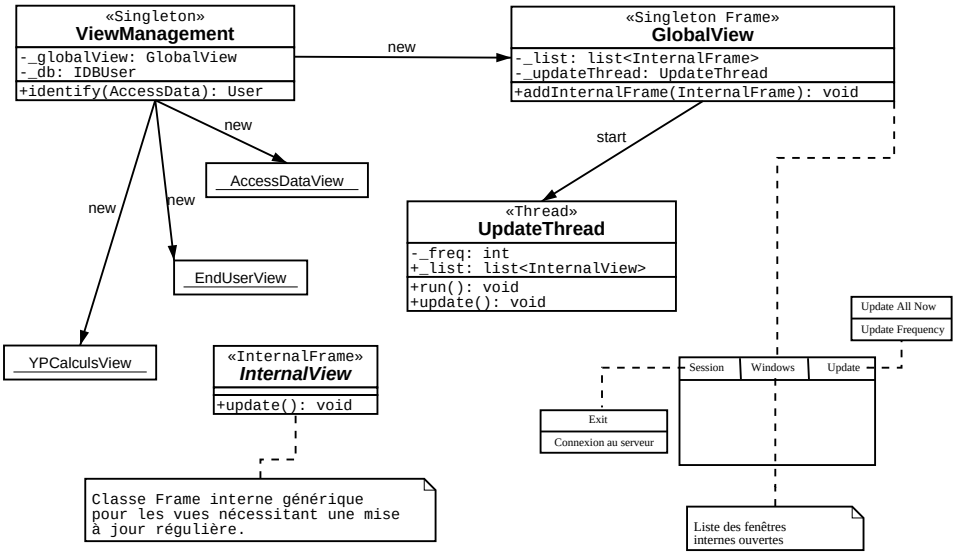

FIG. 3.18 – Diagramme U.M.L. des classes générales de vue et de gestion.

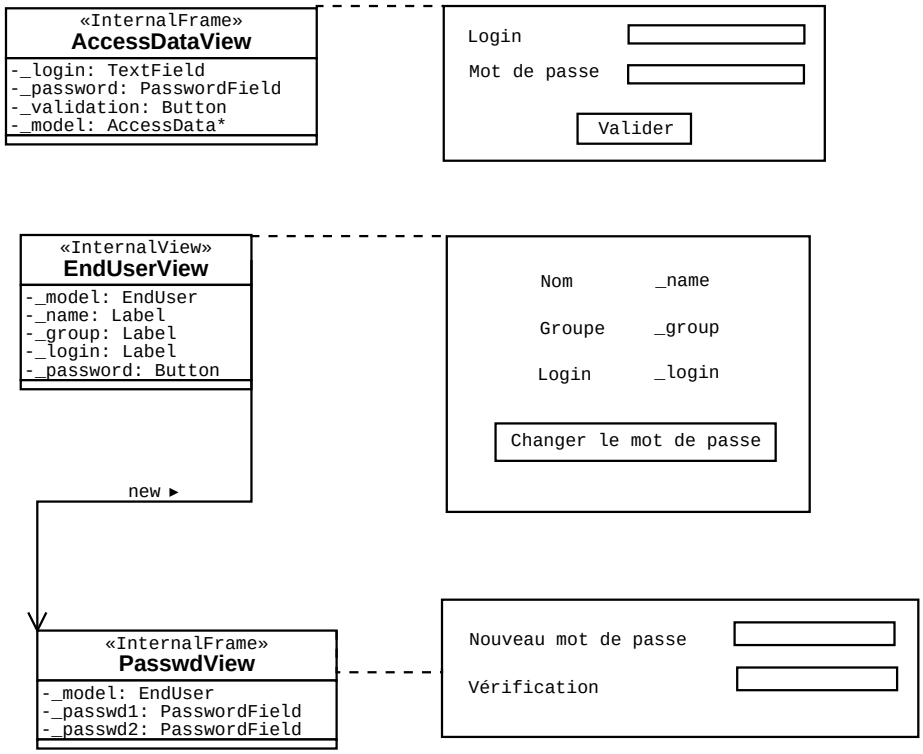

FIG. 3.19 – Diagramme U.M.L. des classes graphiques liées à l'objet utilisateur.

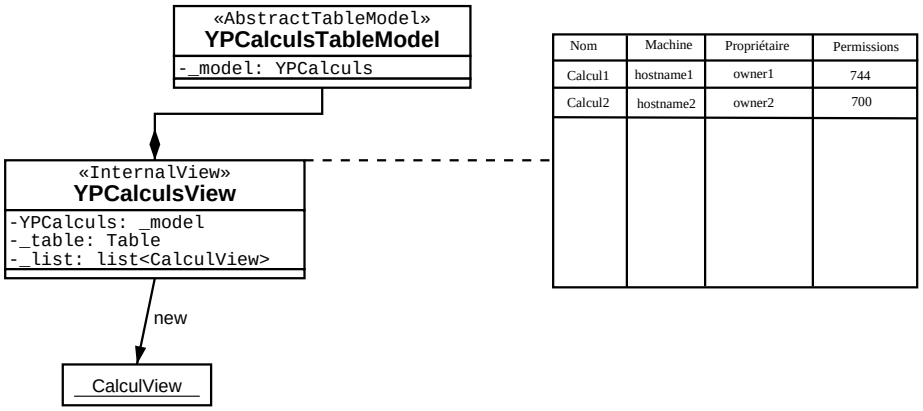

FIG. 3.20 – Diagramme U.M.L. des classes graphiques liées aux pages jaunes des exécutions.

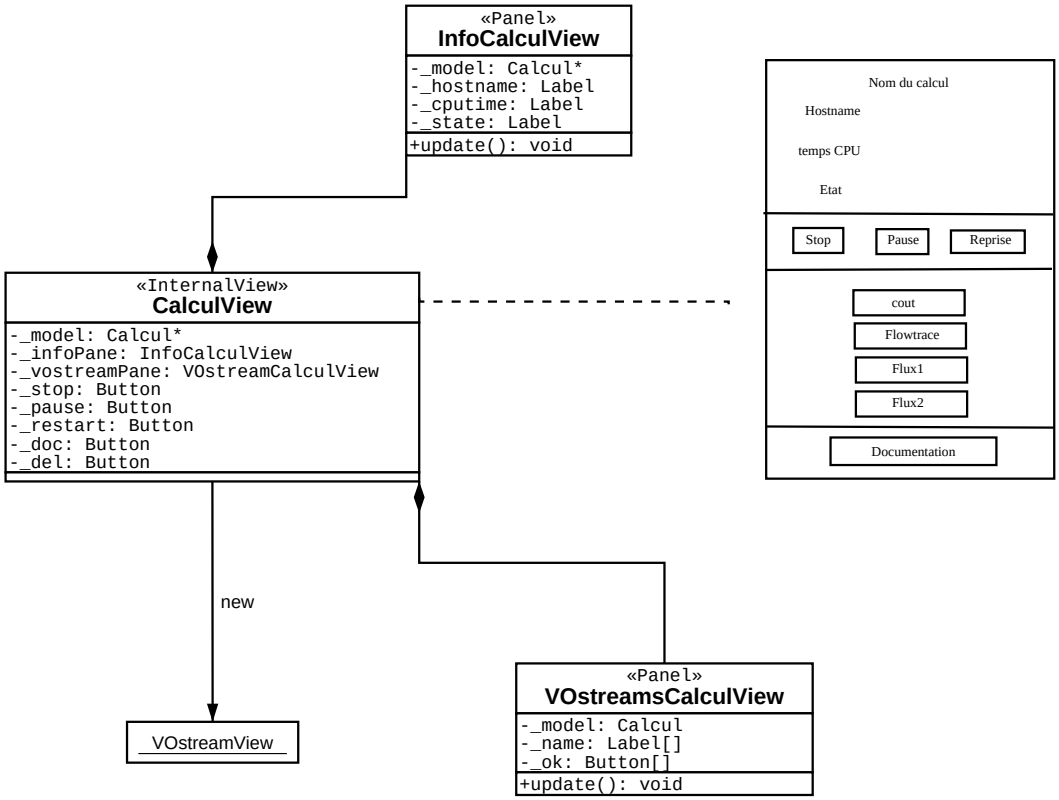

FIG. 3.21 – Diagramme U.M.L. des classes graphiques liées à une exécution.

Enfin, le diagramme 3.22 correspond aux vues des données elles-mêmes.

3.6.5 Module Code

Le module *Code* a déjà été partiellement détaillé lors de la conception préliminaire concernant le module *Execution*. Le rôle de l'objet contrôle *CodeManagement* défini lors de l'analyse est partiellement tenu par l'objet *Process*.

Le diagramme 3.23 illustre la conception de la partie instrumentation du code de calcul.

3.7 Conception détaillée

La conception détaillée consiste à compléter les classes de la conception préliminaire de façon à ce que leurs interfaces soient complètement définies et documentées. Comme les langages de programmation et les protocoles qui vont être utilisés sont connus, on peut être beaucoup plus précis au niveau des diagrammes de classes :

- précision des types de données,
- ajout de nouveaux attributs et de nouvelles méthodes ...

Comme l'illustrent les sections précédentes, l'architecture définit naturellement trois grands sous-systèmes :

- le sous-système code de calcul,
- le sous-système serveur de données,
- le sous-système utilisateur final.

Selon ce schéma, on peut donc diviser la conception détaillée en trois phases :

- conception détaillée des classes du sous-système code de calcul,
- conception détaillée des classes du sous-systèmes serveur de données,

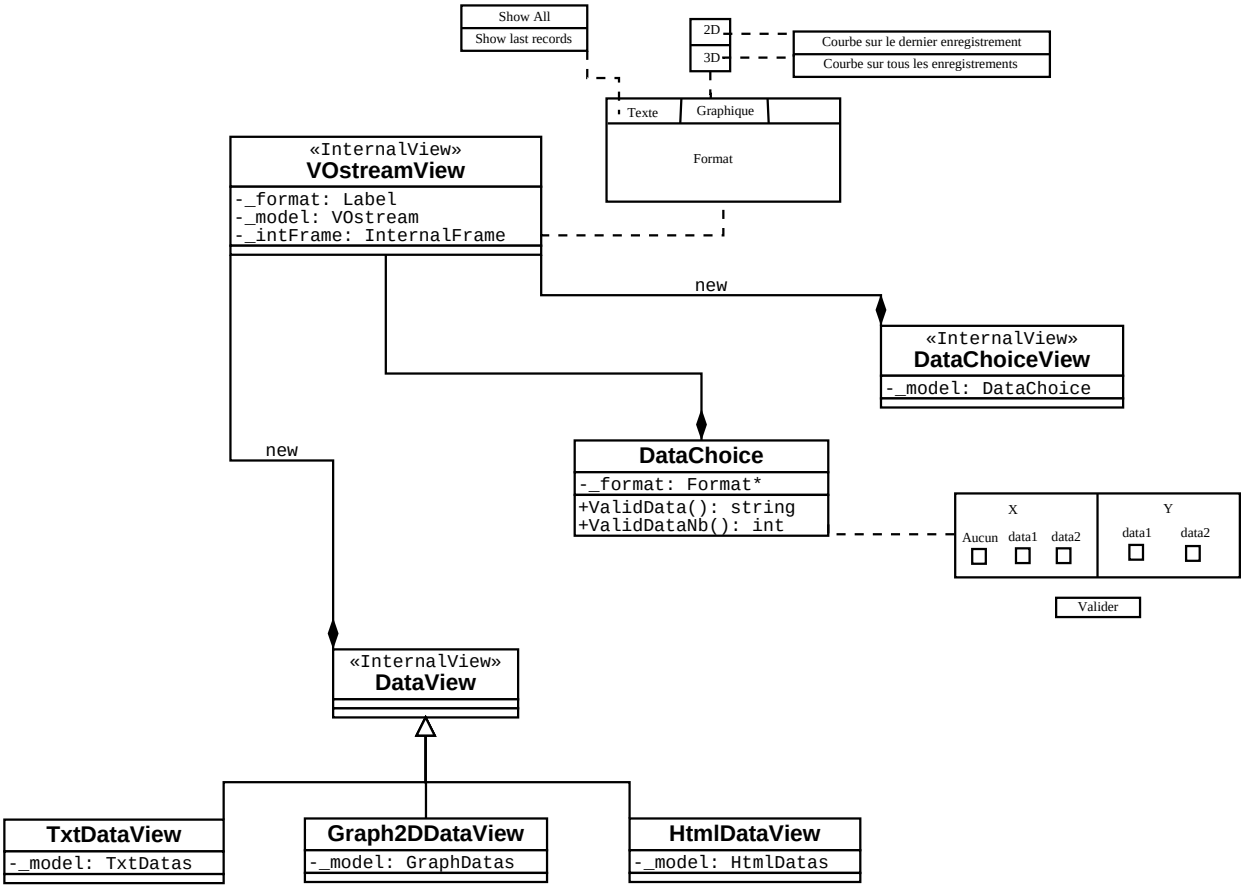

FIG. 3.22 – Diagramme U.M.L. des classes graphiques liées aux données.

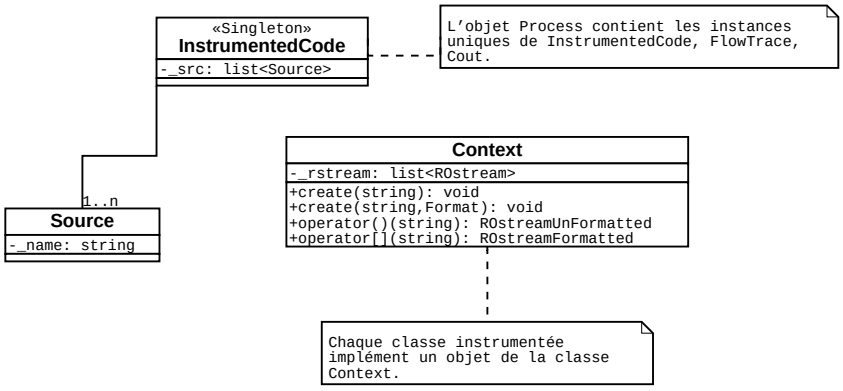

FIG. 3.23 – Diagramme U.M.L. des classes issues de la conception de la partie instrumentation du code de calcul.

– conception détaillée des classes du sous-système utilisateur final.

Pour des raisons de place et de lisibilité seules quelques classes importantes issues de la conception détaillée seront illustrées par des diagrammes UML.

3.7.1 Conception détaillée du sous-système serveur de données

Le langage choisit pour l'implémentation de la partie serveur de données est le C++. De plus, le serveur communique avec les autres sous-systèmes par l'intermédiaire du protocole CORBA. Ces nécessités techniques font apparaître de nouvelles classes nécessaires à la mise en place d'un serveur CORBA ([**?**], [**?**])

Ainsi, il va falloir détailler certains aspects propres à ce protocole :

– Les objets et services CORBA,
– les objets communiquants et leur contrat IDL,
– les fabriques d'objets associées aux objets communiquants.

Classes Corba

Une nouvelle classe *CorbaServer* permet d'initialiser les références CORBA et les services CORBA indispensables au bon fonctionnement du serveur. Ceux-ci sont listés ci-dessous.

L'ORB L'ORB (Object Request Broker) est le noyau de transports de requêtes aux objets distribués. Il intègre notamment le protocole IIOP (Internet Inter-ORB Protocol) permettant l'interconnexion de bus CORBA au dessus de TCP/IP. L'ORB permet de contrôler le comportement du bus, entres autres de créer les autres objets représentants les composantes du bus, ou d'obtenir les références des objets particuliers tels que les différents services CORBA.

Le POA Le POA (Portable Object Adapter) est l'adaptateur d'objet permettant :

– la génération et l'interprétation des références d'objets,
– la connaissance de l'implantation courante associée à un objet,
– la délégation des requêtes aux objets à leur implantation.

Le Naming Service Le service de nommage fait partie des services de recherches proposés par le protocole CORBA. Il permet la recherche des objets en fonction de noms symboliques leur ayant été associés. Le serveur de nom est exécuté en tâche de fond sur le serveur. La référence du service est obtenue par l'appel d'une méthode d'initialisation sur le nom particulier "NameService".

Les noms sont constitués d'une séquence ordonnée de composants, structurés par un graphe de contexte de nommage. Les contextes de noms utilisés dans le cadre du projet GTOOCS sont explicités un peu plus loin, lors de la description des objets qui y sont référencés.

L'Event Service Par défaut, la coopération des objets CORBA est réalisé selon un mode de communication client/serveur synchrone. Toutefois, il existe un ensemble de services assurant des communications asynchrones, dont le service d'évènement. Il permet aux objets de produire des évènements asynchrones à destination d'objets consommateurs à travers des canaux d'évènements.

Le service d'évènements CORBA est, comme le service de nommage, exécuté en tâche de fond sur la machine serveur.

Les envois et réceptions d'évènements se font à travers des canaux d'évènements. Ceux-ci sont eux-mêmes créés par le biais d'une fabrique.

Le diagramme de classe 3.24 illustre cette nouvelle classe du serveur de données.

Objets communiquants

Les objets communiquants sont ceux qui sont accessibles sur le serveur à partir d'une application cliente. Cette définition s'applique à certaines classes issues de la conception préliminaire :

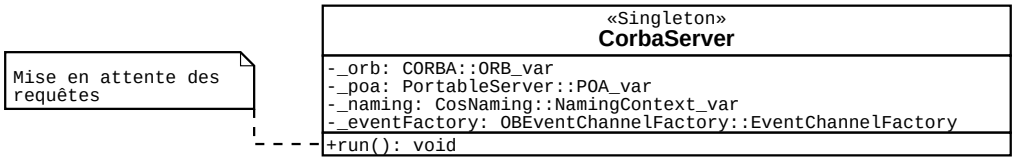

FIG. 3.24 – Diagramme des classes Corba du serveur.

module *DBUser* : classes *UserManagement* et *User*,

module *Stream* : classes de la hiérarchie *Ostream*, classes *Record* et *Format*,

module *Execution* : classes *Execution*, *YPExecutions, AccessAuthorization* et classes de la hiérarchie *ControlAction*.

Les objets du module *DBUser* sont des objets distribués accessibles par le biais du serveur de base de données. La classe interfaçant la base de données doit donc implémenter les méthodes et attributs nécessaires à l'accession à cette base. C'est la classe *UserDataBase* qui réalise cette interface.

Les autres objets communiquants ne se manipulent que par le biais de leur interface, c'est-à-dire par l'intermédiaire des services qu'ils proposent. Ces fonctionnalités sont décrites à l'aide d'un langage propre à CORBA : l'IDL (Interface Definition Language).

La description des interfaces avec l'IDL permet de définir un contrat entre les clients et les serveurs.

Il faut donc documenter de façon précise les interfaces des objets communiquants comme il est illustré dans le diagramme 3.25.

Fabriques associées aux objets communiquants

Les objets communiquants sont soit créés par le serveur CORBA, soit créés par les applications clientes du serveur, c'est-à-dire le sous-système Code de Calcul ou le sous-système IHM. Dans ce dernier cas, ils sont instanciés par le biais de fabriques d'objets référencées sur le serveur. La conception de ces classes est basée sur le patron architectural factory. Les références des objets factory sont publiques et permettent aux sous-systèmes clients de créer sur le serveur autant d'objets distribués que nécessaires. Ces objets ne peuvent être créés et détruits que par le biais des fabriques.

Les classes d'objets nécessitant la mise en oeuvre d'une factory sont les suivantes :
- Classe *Execution*,
- Classe *AccessAuthorization*,
- Classe *ControlAction*,
- Classe *Ostream*.

On obtient ainsi de nouvelles classes issues de la conception détaillée. Leurs interfaces sont données par les contrats IDL illustrés par le diagramme 3.26.

Servant des objets communiquants

Tous les objets communiquant définis par une interface IDL sont implémentés sur le serveur par le biais de servants. Ceux-ci sont dérivés du squelette généré par le compilateur IDL et implémentent ainsi nécessairement chacun des services présents dans le contrat IDL.

Ces servants sont liés aux objets communiquants par le biais du POA, et ce sont eux qui réalisent les requêtes des clients sur les objets CORBA.

De nouvelles classes apparaissent donc, dont le nom, par convention, reprend celui de l'objet CORBA accompagné du suffixe *"_impl"*.

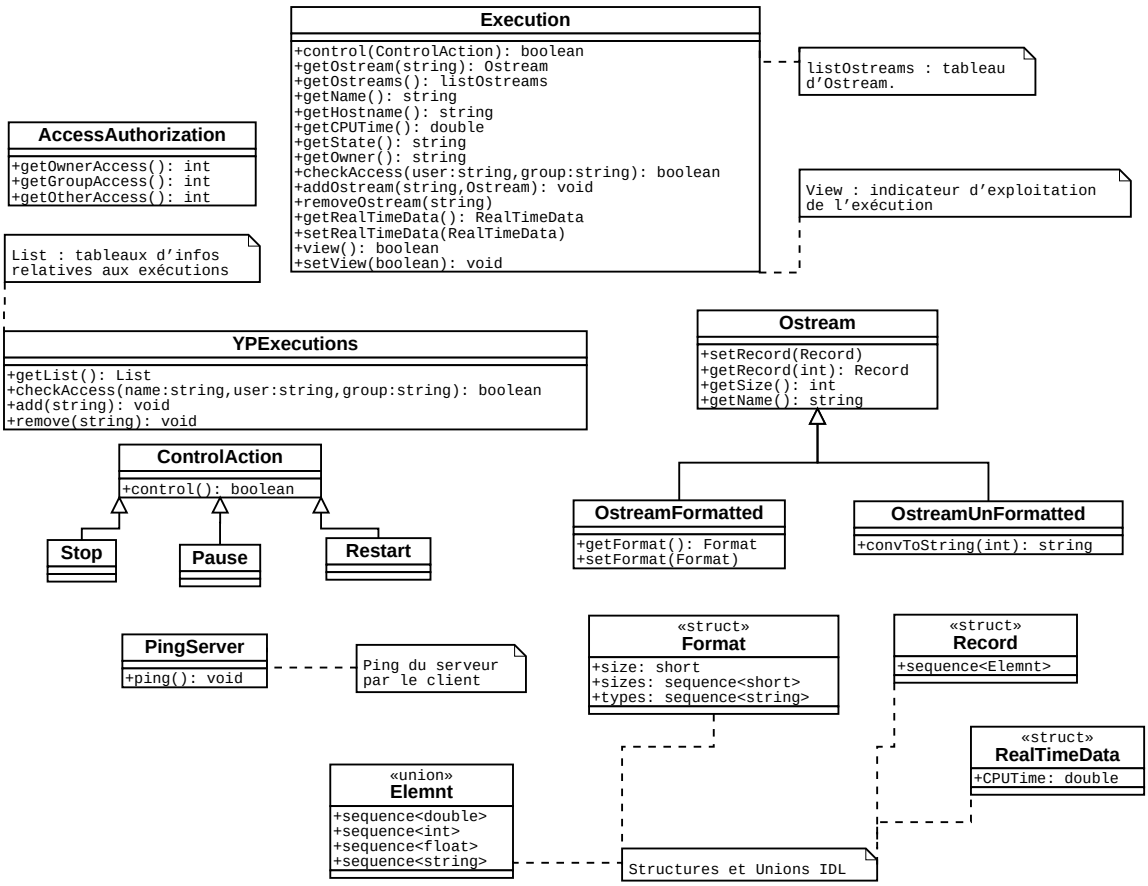

FIG. 3.25 – Diagramme des interfaces IDL des objets communiquants.

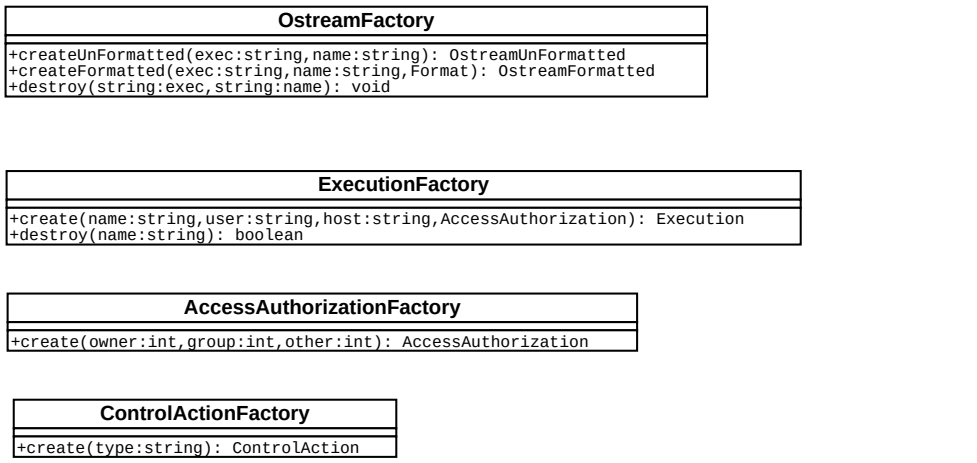

FIG. 3.26 – Diagramme des interfaces IDL des fabriques d'objets communiquants.

Gestion du nommage, gestion des canaux d'évènements

Gestion du nommage Les objets communiquants de type *Ostream* et *Execution* sont référencés sur le service de nommage selon le schéma 3.27. Si on se réfère à la structure des fichiers sur un système Unix, on a les analogies suivantes :

- les rectangles correspondent aux contextes de nom, qui peuvent être interprêtés comme des répertoires,
- les ovales correspondent aux noms, qui peuvent être interprêtés comme des fichiers.

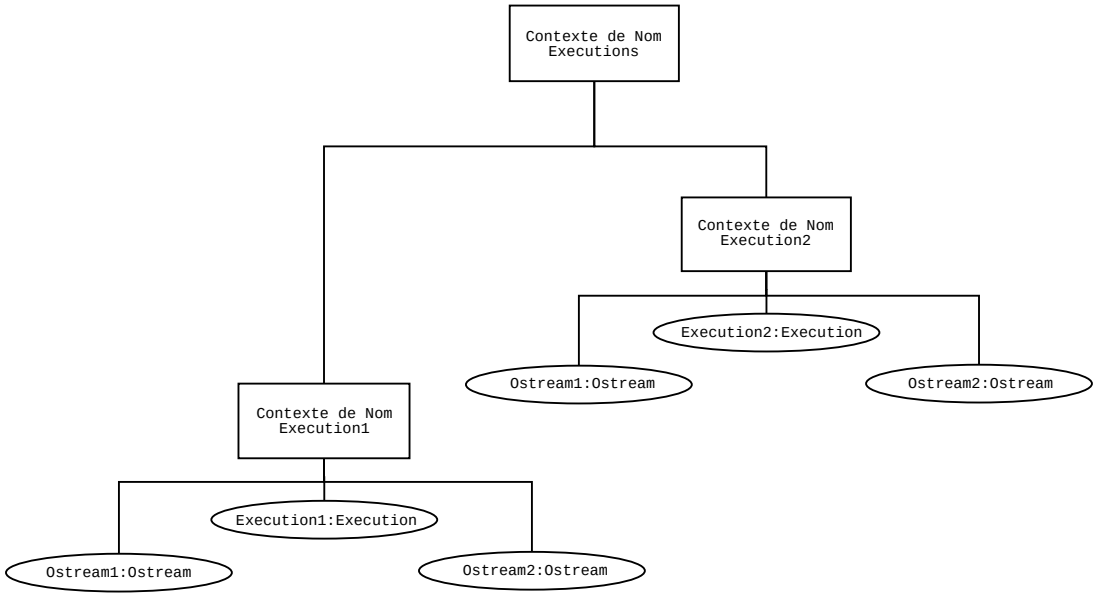

FIG. 3.27 – Gestion des contextes de nom relatifs aux données et exécutions.

De façon analogue, les fabriques des objets communiquants, les pages jaunes des exécutions et l'objet de ping du serveur sont référencés sur le service de nommage sous le contexte de nom racine selon le schéma 3.28.

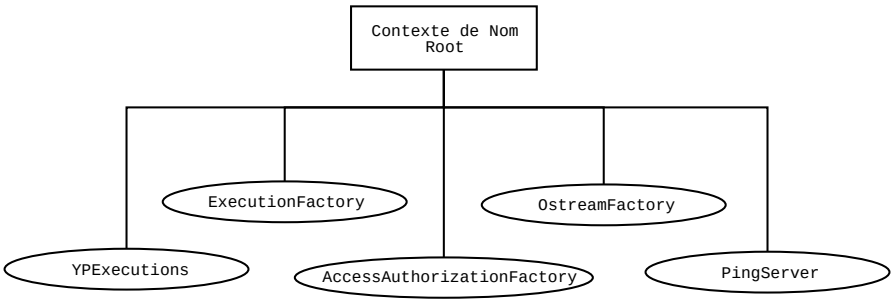

FIG. 3.28 – Gestion du contexte de nom racine.

Gestion des canaux d'évènements La mise à jour de certaines des données entre le serveur et les applications clientes (IHM, codes de calcul) est réalisée par l'intermédiaire des canaux d'évènements. Lors de la création d'un nouvel objet communiquant de type *Execution*, deux nouveaux canaux d'évènements sont créés par le biais de la fabrique.

Ainsi, les données temps réel relatives à un objet *Execution*, ainsi que les commandes de contrôle de cette exécution, transitent à travers les canaux d'évènements propres à cette exécution.

3.7.2 Conception détaillée du sous-système code de calcul

Le langage choisit pour l'implémentation de la partie calcul du logiciel est le C++. De plus, le code de calcul doit communiquer avec le serveur de données par l'intermédiaire du protocole CORBA. Ces nécessités techniques font apparaître de nouvelles classes, notamment pour l'accès aux références des objets CORBA tels que l'ORB, le NamingService ou la fabrique des canaux d'évènements. D'autre part, les classes instrumentales décrites en 3.6.5 sont complétées par de nouvelles classes permettant notamment la gestion des évènements reçus et envoyés par le client.

Classes Corba

La classe singleton *CorbaClient* permet d'initialiser les références de ces objets CORBA. Son interface est illustrée par le diagramme 3.29.

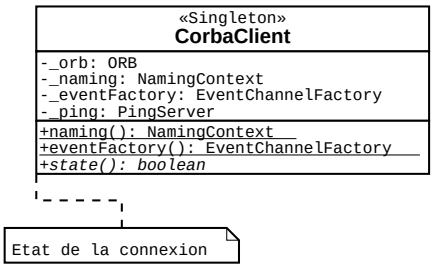

FIG. 3.29 – Classes Corba du client.

D'autre part, le code de calcul va être amené à créer des objets distribués sur le serveur (objets *Ostream*, *AccessAuthorization* ...). Il faut donc prévoir des classes interfaçant les fabriques d'objets du serveur. Ces classes sont nommés *RExecutionFactory*, *RAccessAuthorizationFactory* et *ROstreamFactory*, le préfixe "R" signifiant "Remote".

Classes instrumentales

Les classes permettant l'instrumentation des codes de calcul déjà définies sont complétées par de nouvelles classes, les unes à vocation secondaire (sous objets des classes présentées), les autres permettant la gestion de l'envoi et de la réception des évènements temps réels.

Ces classes sont des threads qui permettent d'accéder instantanément à un évènements dès que celui-ci est communiqué au canal concerné.

Ces classes de gestion d'évènements sont illustrées par le diagramme 3.30.

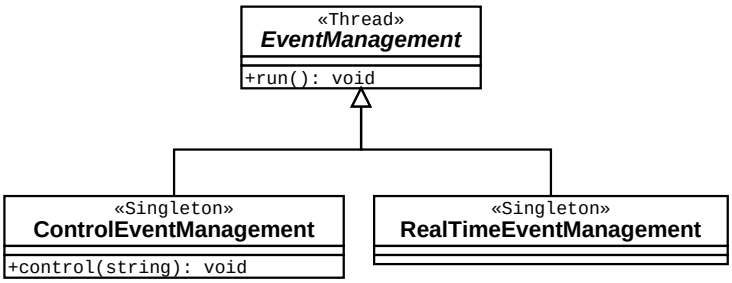

FIG. 3.30 – Diagramme des classes de gestion d'évènements.

3.7.3 Conception détaillée du sous-système utilisateur final

Le langage choisit pour la partie IHM est Java. L'IHM doit pouvoir communiquer avec le serveur CORBA, mais aussi avec le serveur de la base de données utilisateurs. Ainsi, les classes interfaçant

la connexion avec le serveur Mysql (notamment la classe *UserDataBase*) doivent également être développées en Java.

Pour la communication avec le serveur CORBA, l'interface de la classe *CorbaClient* défini pour le sous-système code de calcul en 3.7.2 est réutilisée.

La conception des autres classes de l'IHM a été suffisamment détaillée dans les parties précédentes, les classes modèles dans l'ensemble de la section 3.6 et les classes de vue dans la sous-section 3.6.4. Par souci de clarté du document, nous ne les expliciterons pas ici.

3.8 Implémentation et tests

3.8.1 Choix techniques pour l'implémentation

Les choix techniques à effectuer concernent les compilateurs, le serveur de base de données, les API de connexions à ce serveur, les librairies graphiques, l'implémentation de l'ORB.

Les contraintes relatives à ces choix techniques imposent d'utiliser des outils répondant aux exeigences suivantes :

- multi-OS,
- libres et gratuits,
- stables et pérennes.

Compilateurs

Le choix des compilateurs reste ouvert car l'implémentation doit impérativement suivre la norme des langages utilisés. Dans un premier temps, les programmes sont compilés grâce aux compilateurs GNU et intel pour le C++, et le SDK de SUN pour Java.

Concernant la partie fortran, le compilateur choisi est le compilateur intel.

Serveur de base de données

Un des serveurs de base de données répondant aux contraintes évoquées est le serveur Mysql.

API de connexion à la base de données

Le SDK de SUN intègre une API de connexion à un serveur SQL pour le langage Java. Concernant le C++, il existe une API nommé Mysql++ permettant de communiquer avec le serveur de base de données MySQL.

Librairies graphiques

Les API du SDK de SUN ne comprennent pas de classes de traçage de courbes évolués, ce qui est indispensable au bon fonctionnement de l'interface graphique du projet. Cependant, il existe un certain nombre de solutions pour résoudre ce problème. Nous avons choisi d'utiliser l'API jfreechart pour la visualisation en java des courbes 2D.

D'autre part, des outils de visualisation de type gnuplot ou AVS peuvent être connectés à l'interface graphique (voir à ce sujet les itérations du processus de conception en 3.9).

Implémentation de l'ORB

Le choix de l'ORB doit, outre les contraintes évoquées ci-dessus, répondre à l'exigence d'un mapping à la fois Java et C++. Cela peut-être réalisé par l'utilisation de deux ORB, la norme imposant une compatibilité entre ORB, ou par un ORB proposant les deux mapping. Cette solution, la plus simple, est celle qui est pour l'instant retenue. Elle dépend de l'évolution des implémentations, tant au niveau des fonctionnalités disponibles, qu'au niveau de la politique de déploiement des construteurs.

L'implémentation retenue est, pour le moment, celle de Orbacus.

3.8.2 Modèle d'implémentation : diagramme de composants

Les diagrammes de composants présentés ci-dessous permettent de décrire l'architecture physique et statique de l'application en terme de modules, c'est-à-dire à partir de fichiers sources, librairies, exécutables, ...

Les dépendances entre composants aident ainsi à identifier les contraintes de compilation, et mettent en évidence la réutilisation de ces composants.

Le diagramme 3.31 montre la structure physique des modules concernant les entités distribuées.

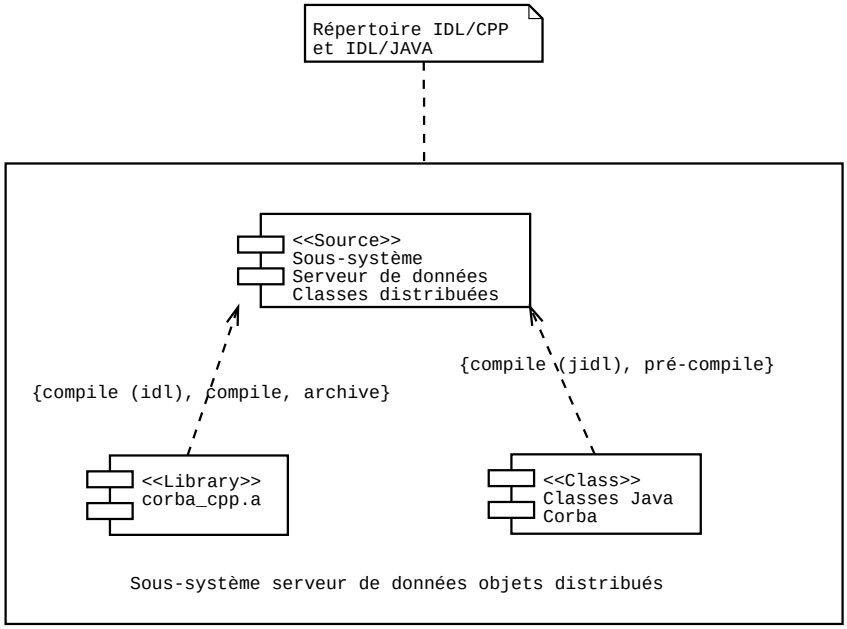

FIG. 3.31 – Diagramme de composants des entités relatives aux objets distribués.

Le diagramme 3.32 illustre l'architecture des fichiers de la base de données des utilisateurs.

Le diagramme 3.33 concerne les fichiers utilisés pour les programmes clients.

Les schémas 3.34 indiquent les librairies extérieures nécessairent au bon fonctionnement du logiciel.

Le diagramme 3.35 illustre les dépendances pour la génération de l'exécutable de la partie serveur. Il utilise les structures qui ont été présentées ci-dessus.

Le schéma 3.36 concerne les dépendances pour les codes de calcul clients.

Enfin, le diagramme 3.37 présente les modules nécessaires à l'exécution de l'interface graphique.

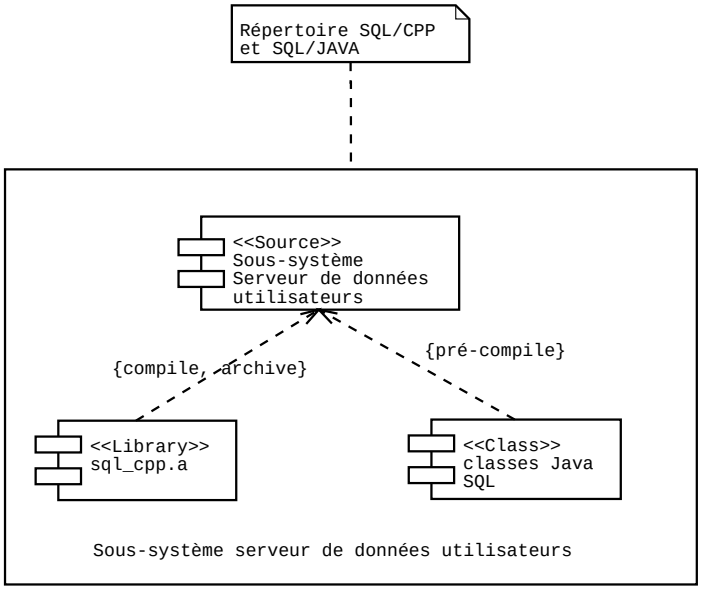

FIG. 3.32 – Diagramme de composants relatif à la base de données utilisateurs.

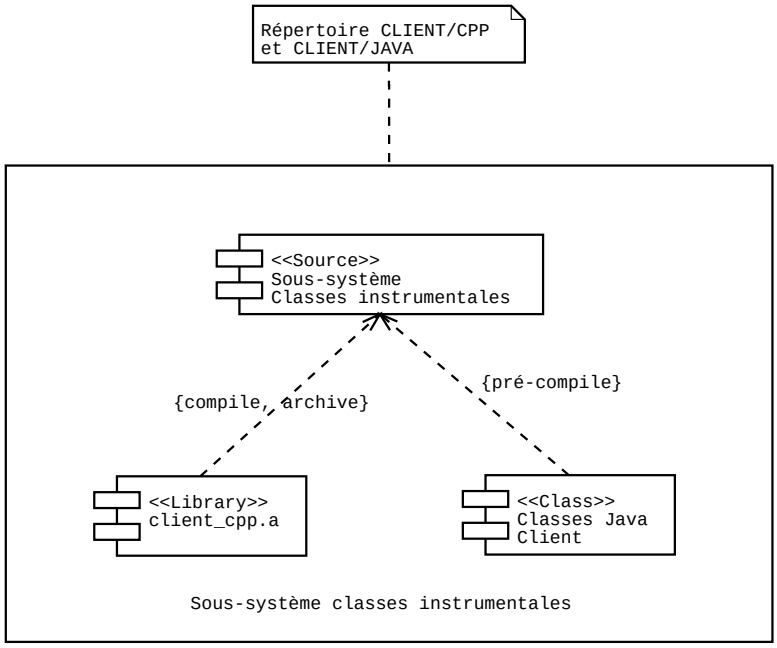

FIG. 3.33 – Diagramme de composants relatif aux programmes clients.

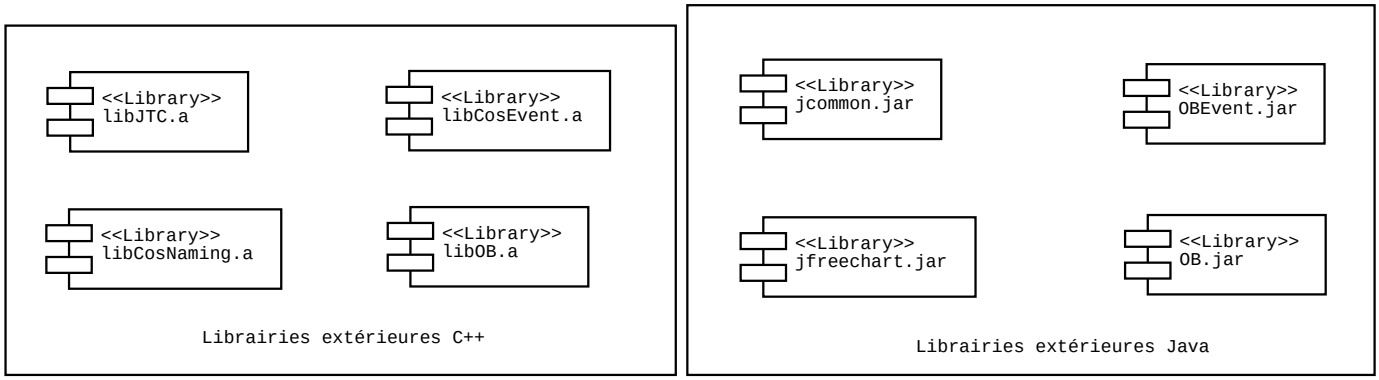

FIG. 3.34 – Diagramme de composants relatif aux librairies extérieures.

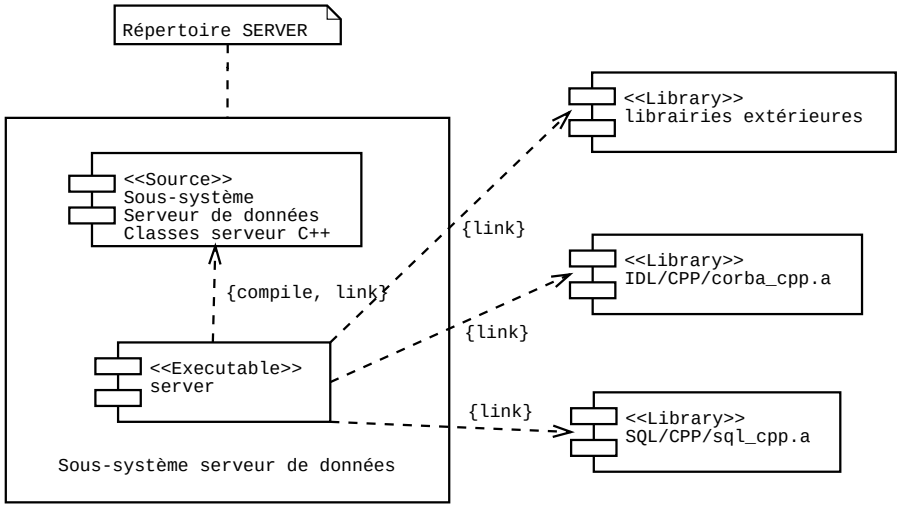

FIG. 3.35 – Diagramme de composants relatif à l'exécutable du serveur.

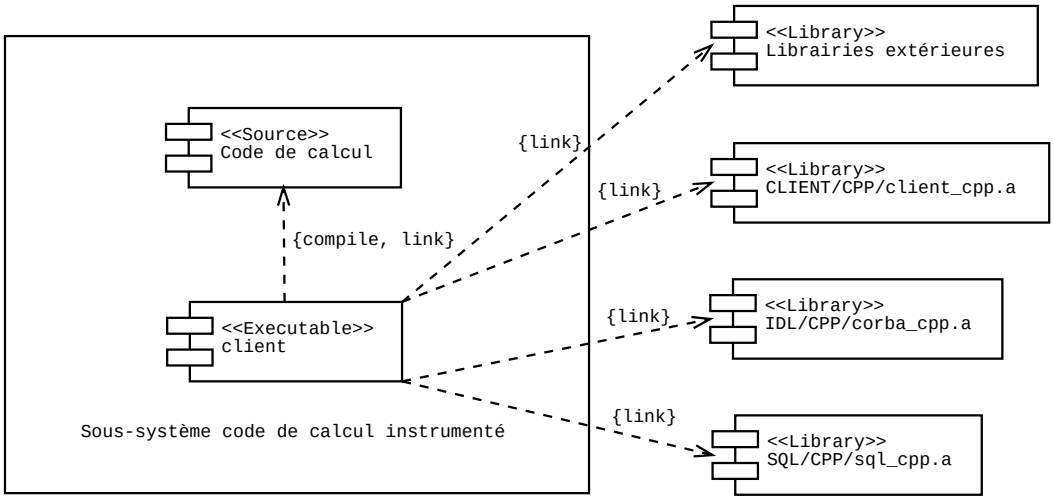

FIG. 3.36 – Diagramme de composants relatif aux exécutables des codes de calcul.

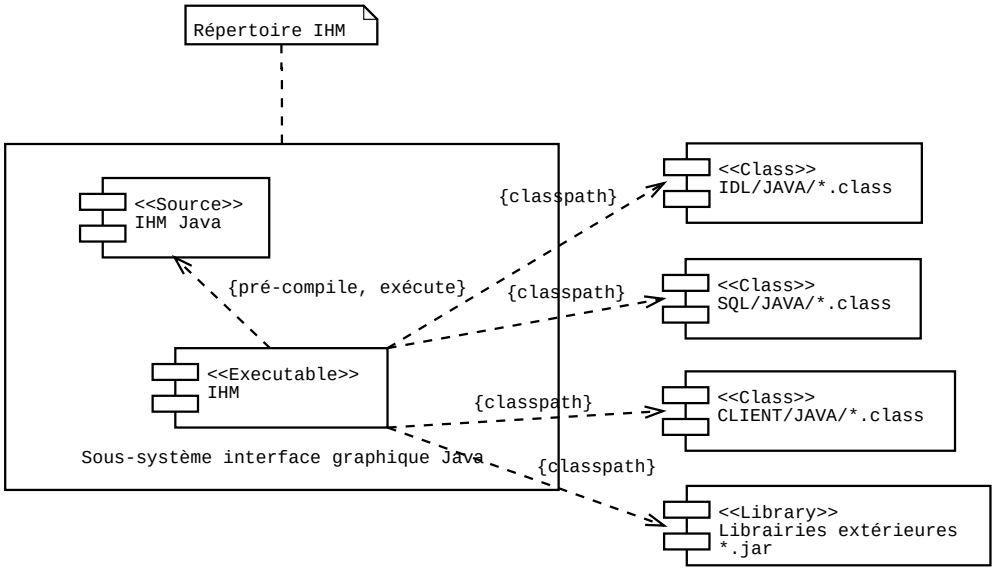

FIG. 3.37 – Diagramme de composants relatif à l'exécution de l'interface graphiquel.

3.8.3 Tests

L'enchaînement des tests s'attache à la vérification des résultats de l'implémentation en testant chaque construction, aussi bien les constructions internes et intermédiaires que les versions finales du système.

La phase de tests est donc essentielle. Elle se décompose en tests unitaires et tests d'intégration ou de déploiement.

Tests unitaires

Le propos de cette activité est de tester les composants implémentés en tant qu'unités individuelles. Les tests unitaires doivent donc être fréquents. Ils doivent comporter des tests de structure qui permettent de vérifier le fonctionnement interne d'un composant, et pour lesquels il faut s'assurer que tout le code est bien testé, chaque instruction devant être exécutée au moins une fois.

Dans le cas de notre système, il faut par exemple s'assurer qu'un code instrumenté peut s'exécuter en standalone sans aucune contrainte particulière, et évidement sans aucune perte de données.

Tests d'intégration

Les cas d'utilisation sont la base des tests d'intégration. Il s'agit de vérifier que les fonctionnalités attendues et décrites par les cas d'utilisation sont effctivement implémentées.

Les tests de déploiements ont donc été réalisés à partir des diagrammes de séquence de chacun des cas d'utilisation décrits en 3.3. On est ainsi assuré d'avoir bien répondu aux attentes exprimés lors de l'expression des besoins.

3.9 Itération du processus

Un système logiciel passe par un certain nombre de cycles de développement pendant sa durée de vie. Chacun de ces cycles donne lieu à une nouvelle version du produit. Le processus de développement d'un logiciel est donc incrémental. Ce point de vue, établi dès le départ, permet de prendre en compte, au fur et à mesure de la vie du système, de nouvelles fonctionnalités.

3.9.1 Scénarii secondaires des cas d'utilisation

La nouvelle itération du processus d'élaboration du logiciel GTOOCS doit prendre en compte les scénarii secondaires des cas d'utilisation, c'est-à-dire les scénarii d'erreur et d'exception. Il faut maintenant intégrer un comportement du logiciel pour le cas où les différents pas du scénario nominal ne peuvent être réalisés.

3.9.2 Nouvelles fonctionnalités

La prochaine version du logiciel doit intégrer de nouvelles fonctionnalités. L'expression de ces nouveaux besoins se décompose en plusieurs fonctions :

- la possibilité de relire les données d'une exécution déjà terminée,
- la possibilité de modifier les données d'une exécution (de type *Istream* en complément du type *Ostream*),
- l'accès à des outils de visualisation à partir de l'interface graphique (AVS, gnuplot, VTK ...),
- la mise en oeuvre d'une bibliothèque d'alerte qui permettra, par un moyen défini (mail, évènement sur l'interface graphique ...), de prévenir l'utilisateur d'un évènement (divergence d'un calcul ...).

D'autre part, l'architecture du logiciel doit évoluer vers la mise en place d'une solution multi-serveurs qui permettra d'optimiser les performances de transfert de données entre des machines lorsque la machine de calcul est très éloignée du serveur.

Enfin, l'optimisation du timing de transfert de données améliorera les performances des codes instrumentés.

Deuxième partie

De la conception à la programmation

Chapitre 4

Application à la résolution d'un problème elliptique

Introduction

Ce document accompagne le rapport sur la programmation orientée objet pour le calcul scientifique. Il en constitue le premier exemple d'application. Il contient le guide d'utilisation d'un programme de résolution du problème elliptique suivant :

$$-\nu\Delta u + \alpha u = f \tag{4.1}$$

discrétisé par éléments finis, par différences finis et bientôt par volumes finis. Ce problème est posé dans Ω un domaine borné de $\mathbb{R}^2$. Enfin nous supposerons que la solution u est donnée sur une partie Γ du bord $\partial\Omega$ de Ω.

Le langage utilisé pour cet exemple est le C++.

Dans les paragraphes 4.1 et 4.2 nous décrirons la discrétisation du problème (4.1) en éléments finis et en différences finis, respectivement. La mise en œuvre fera l'objet du paragraphe 4.3.

4.1 Discrétisation par éléments finis

On considère un ouvert polygonal borné Ω de $\mathbb{R}^2$. On cherche la solution du problème (4.1) suivant :

$$\left\{ \begin{array}{rcll} -\nu\Delta u + \alpha u & = & f, & \text{dans } \Omega \\ u & = & 0, & \text{sur } \Gamma \end{array} \right.$$

où ν et α sont deux nombres réels donnés ($\nu > 0$ et $\alpha \geq 0$). On suppose que f est une fonction de carré sommable. Pour discrétiser ce problème par une méthode d'éléments finis on considère sa formulation variationnelle suivante :

$$\left\{ \begin{array}{c} \text{On cherche } u \text{ dans } V \text{ telle que pour tout } v \text{ dans } V \text{ on ait} \\ \nu\left(\nabla u, \nabla v\right) + \alpha\left(u, v\right) = \left(f, v\right), \end{array} \right. \tag{4.2}$$

où $(\cdot,\cdot)$ représente le produit scalaire dans $L^2(\Omega)$ et V l'espace fonctionnelle

$$V = \left\{ v \in H^1(\Omega), \text{ telle que } v = 0 \text{ sur } \Gamma. \right\}$$

On notera également $H = L^2(\Omega)$.

La discrétisation par la méthode d'éléments finis repose sur un maillage $\mathcal{T}_h$ de Ω. On supposera que celui-ci est composé uniquement de triangles ou de quadrangles (maillage ou triangulation homogène, cette restriction sera levée dans les versions ultérieures). L'indice h représente la taille moyenne d'un élément.

La mise en œuvre proposée dans ce travail repose sur un formalisme unique quelque soit le mode de discrétisation. Nous décrivons le cas des éléments finis. On note $\mathcal{P}$ l'opérateur qui à tout f de H lui fait correspondre u de V solution de (4.2). On note V_h le sous-espace de dimension finie de V défini par :

$$V_h = \left\{ v_h \in \mathcal{C}^0(\Omega) \text{ telle que } v_{h|_\tau} \in \mathcal{P}_k(\tau) \text{ pour tout } \tau \in \mathcal{T}_h \text{ et telle que } v_h = 0 \text{ sur } \Gamma \right\}$$

où $\mathcal{P}_k(\tau)$ est l'ensemble des polynômes de degré inférieur ou égal à k sur τ. A l'espace d'éléments finis V_h est associé un ensemble de degrés de liberté. Il possède une base dont on nommera les éléments ϕ_i pour i allant de 1 à N_h la dimension de V_h. Classiquement, on définit le problème discret comme suit :

$$\left\{ \begin{array}{c} \text{On cherche } u_h \text{ dans } V_h \text{ telle que pour tout } v_h \text{ dans } V_h \text{ on ait} \\ \nu\left(\nabla u_h, \nabla v_h\right) + \alpha\left(u_h, v_h\right) = \left(f, v_h\right). \end{array} \right. \tag{4.3}$$

On définit les applications suivantes entre les espaces discrets et les espaces continus.

$$p_h : V \mapsto V_h$$

L'application qui à tout v de V fait correspondre v_h de V_h est définie par :

$$v_h = \sum_{i=1}^{N_h} v_i \phi_i$$

où v_i représente la valeur de v au i-ième degré de liberté associé à la fonction ϕ_i.

De façon analogue, on définit H_h un sous-espace de dimension finie de H.

$$H_h = \left\{ v_h \in \mathcal{C}^0(\Omega) \text{ telle que } v_{h|_\tau} \in \mathcal{P}_{k'}(\tau) \text{ pour tout } \tau \in \mathcal{T}_h \right\}$$

Le degré des polynômes k' peut être différent de celui de l'espace V_h. On définit l'opérateur $q_h : H \mapsto H_h$ de façon analogue à la définition de l'opérateur p_h. On note ψ_i pour i allant de 1 à M_h la base de H_h (M_h est le cardinal de H_h). On a donc le diagramme suivant.

$$\begin{array}{ccc} H_h & \xrightarrow{\mathcal{P}_h} & V_h \\ p_h \uparrow & & \downarrow q_h^* \\ H & \xrightarrow{\mathcal{P}} & V \\ p_h^* \uparrow & & \downarrow q_h \\ H_h & \xrightarrow{\mathcal{P}_h} & V_h \end{array}$$

L'opérateur discret $\mathcal{P}_h$ est donc défini par :

$$\mathcal{P}_h = q_h \circ \mathcal{P} \circ p_h^*.$$

Cette définition de l'opérateur discret est équivalente à la première qui a été donnée. Ici les espaces discrets sont des sous-espaces des espaces continus, les opérateurs p_h^* et q_h^* sont des injections.

Pour notre exemple, H et H_h sont les espaces dans lesquels vivent respectivement les seconds membres continu et discret. Dans notre cas, ce second membre sera donné analytiquement (voir le paragraphe 4.3). Il ne serait donc pas utile de définir l'espace H_h, mais par compatibilité avec les autres méthodes de discrétisation, nous conserverons la discrétisation de la fonction source f et l'on choisira $k' = k$ afin de ne gérer qu'un seul type d'éléments finis.

Il reste à expliciter la forme matricielle du problème discret $\mathcal{P}_h$. On utilise une formule d'intégration numérique pour le calcul des deux opérateurs discrets de notre formulation :

$$\begin{aligned} \left(\nabla u_h, \nabla \phi_i\right) &= \int_\Omega \nabla u_h \nabla \phi_i d\Omega = \sum_{\tau \in \mathcal{T}_h} \int_\tau \nabla u_h \nabla \phi_i d\Omega = \sum_{\tau \in \mathcal{T}_h} \sum_{j=1}^{N_h} u_j \int_\tau \nabla \phi_j \nabla \phi_i d\Omega \\ &= \sum_{\tau \in \mathcal{T}_h} \sum_{j=1}^{N_h} u_j \sum_{k=1}^{K} c_k \nabla \phi_j(x_{k,\tau}) \nabla \phi_i(x_{k,\tau}) \end{aligned}$$

où $x_{k,\tau}$ représente les coordonnées du k-ième point d'intégration de l'élément τ, c_k est le coefficient associé et K leur nombre. On obtient des formules analogues pour l'opérateur identité. On construit ainsi une matrice A donc les coefficients sont :

$$A_{i,j} = \sum_{\tau \in \mathcal{T}_h} \sum_{k=1}^{K} c_k \nabla\phi_j(x_{k,\tau}) \nabla\phi_i(x_{k,\tau}).$$

On appelle coefficient élémentaire la contribution d'un élément τ au coefficient, soit :

$$a_{i,j}^{\tau} = \sum_{k=1}^{K} c_k \nabla\phi_j(x_{k,\tau}) \nabla\phi_i(x_{k,\tau}).$$

Le calcul de ce coefficient élémentaire subit encore une transformation. Au lieu de le calculer directement sur l'élément courant τ, on passe par l'intermédiaire d'un élément de référence $\hat{\tau}$ et de l'application $\mathcal{F}$ qui transforme l'élément de référence en l'élément courant τ. Plusieurs façon de procéder sont encore possible, nous avons choisit l'interprétation iso-paramétrique de cette application. Cette méthode utilise les fonctions de bases $(\phi_i)_{i=1,\ldots,n+1}$.

$$\mathcal{F}\begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} = \sum_{i=1}^{k+1} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \hat{\phi}_i \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix}$$

Cette définition est aussi valable pour les maillages par des quadrangles et peut être généralisée dans la dimension 3. La matrice jacobienne de $\mathcal{F}$ est donnée par :

$$\mathcal{J} = \sum_i \begin{pmatrix} x_i \frac{\partial}{\partial x}\hat{\phi}_i & x_i \frac{\partial}{\partial y}\hat{\phi}_i \\ y_i \frac{\partial}{\partial x}\hat{\phi}_i & y_i \frac{\partial}{\partial y}\hat{\phi}_i \end{pmatrix}$$

Cette transformation peut être également utilisée pour les éléments finis curvilignes.

Les vecteurs associés respectivement à la solution u_h et au second membre f_h sont formés des v_i et des f_i.

4.2 Discrétisation par différences finis

Sur une grille régulière en dimension 2, on peut définir la discrétisation par différences finies comme le produit extérieur des discrétisations en dimension 1 dans chaque direction. On décrit donc le formalisme unique pour les différences finies en dimension 1, puis on étendra à la dimension 2.

En dimension 1, sur $[0, 1]$ par exemple, le problème (4.1) s'écrit

$$-\nu \frac{d^2}{dx^2}(u) + \alpha u = f$$

avec $u(0) = 0$ comme condition aux limites.

Il s'agit maintenant de construire un diagramme équivalent de celui obtenu en éléments finis. Pour cela on introduit les notations suivantes en supposant que l'on travaille dans le segment unité. Soit N un entier positif, on note $h = 1/N$ le pas de discrétisation. Les nœuds de la grille de calcul sont, pour i variant entre 0 et N : $x_i = ih$. Les domaines de contrôle ω_i sont des segments de taille h centrés sur le point x_i :

$$\omega_i = \left[(i - \frac{1}{2})h,\ (i + \frac{1}{2})h\right] \cap [0,\ 1].$$

On définit le vecteur suivant, pour i dans $\{0, \ldots, N\}$:

$$P_i(x) = \left((x - x_i)^2, (x - x_i), 1\right).$$

On cherche une matrice A carrée d'ordre 3 inversible et ne dépendant pas le l'indice i telle que

$$\begin{array}{lcllcl} A^{-1}Y_i \cdot P_i(x_{i+1}) & = & u(x_{i+1}) & = & u_{i+1} \\ A^{-1}Y_i \cdot P_i(x_{i-1}) & = & u(x_{i-1}) & = & u_{i-1} \\ A^{-1}Y_i \cdot P_i(x_i) & = & u(x_i) & = & u_i \end{array}$$

où

$$Y_i = (u_{i+1},\ u_{i-1},\ u_i)^t.$$

Par identification on trouve

$$A^{-1} = \begin{pmatrix} 1/h^2 & 1/h^2 & -2/h^2 \\ 1/h & -1/h & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{et} \quad A = \begin{pmatrix} h^2/2 & h/2 & 1 \\ h^2/2 & -h/2 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Alors pour tout x dans le domaine de contrôle ω_i on obtient :

$$P_h(u)(x) = A^{-1}Y_i \cdot P_i(x).$$

Maintenant si l'on a des valeurs u_i en chaque point de la grille, on peut définir :

$$u_h(x) = A^{-1}Y_i \cdot P_i(x) \text{ pour } x \in \omega_i.$$

En un point i interne on retrouve la forme habituelle du laplacien :

$$\Delta u_i = \frac{1}{h^2}\left(-u_{i-1} + 2u_i - u_{i+1}\right)$$

On obtient de façon analogue les formules d'ordre 2 sur les points du bord. Par exemple pour $i = 0$, la matrice A est bien entendu différente :

$$\begin{array}{lcllcl} A^{-1}Y_0 \cdot P_0(x_2) & = & u(x_2) & = & u_2 \\ A^{-1}Y_0 \cdot P_0(x_1) & = & u(x_1) & = & u_1 \\ A^{-1}Y_0 \cdot P_0(x_0) & = & u(x_0) & = & u_0 \end{array}$$

où

$$Y_0 = (u_2,\ u_1,\ u_0)^t.$$

On trouve

$$A^{-1} = \begin{pmatrix} 1/(2h^2) & -1/h^2 & 1/(2h^2) \\ -1/(2h) & 2/h & -3/(2h) \\ 0 & 0 & 1 \end{pmatrix} \quad \text{et} \quad A = \begin{pmatrix} 4h^2 & 2h & 1 \\ h^2 & h & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Au point d'abscisse 0, le laplacien s'écrit :

$$\Delta u_0 = \frac{1}{h^2}\left(\frac{u_0}{2} - u_1 + \frac{u_2}{2}\right).$$

Au point d'abscisse 1, on a

$$\Delta u_N = \frac{1}{h^2}\left(\frac{u_{N-2}}{2} - u_{N-1} + \frac{u_N}{2}\right).$$

Pour obtenir les formules à l'ordre 4, on procède de la même manière. Il faut cette fois considérer dans l'expression de $P_i(x)$ 5 termes jusqu'à l'ordre 4 et deux cas particuliers à chaque extrémité :

$$\begin{array}{lcll} \Delta u_0 & = & \frac{1}{h^2}(15/4u_0 - 77/6u_1 + 107/6u_2 - 13u_3 + 61/12u_4 - 5/6u_5) & \\ \Delta u_1 & = & \frac{1}{h^2}(5/6u_0 - 5/4u_1 - 1/3u_2 + 7/6u_3 - 1/2u_4 + 1/12u_5) & \\ \Delta u_i & = & \frac{1}{h^2}(-1/12u_{i-2} + 4/3u_{i-1} - 5/2u_i + 4/3u_{i+1} - 1/12u_{i+1}) & \text{pour } 1 < i < N-1 \\ \Delta u_{N-1} & = & \frac{1}{h^2}(1/12u_{N-5} - 1/2u_{N-4} + 7/6u_{N-3} - 1/3u_{N-2} - 5/4u_{N-1} + 5/6u_N) & \\ \Delta u_N & = & \frac{1}{h^2}(-5/6u_{N-5} + 61/12u_{N-4} - 13u_{N-3} + 107/6u_{N-2} - 77/6u_{N-1} + 15/4u_N) & \end{array}$$

On peut maintenant établir le diagramme pour les différences finis. La solution u vit dans le même espace V inclus dans $H^1(\Omega)$, le second membre f dans son dual V' qui contient H^{-1}.

$$\begin{array}{ccc} V_h & \xrightarrow{\mathcal{P}_h} & V_h \\ p_h \uparrow & & \downarrow p_h^* \\ V & \xrightarrow{\mathcal{P}} & V \\ p_h^* \uparrow & & \downarrow p_h \\ V_h & \xrightarrow{\mathcal{P}_h} & V_h \end{array}$$

4.3 Mise en œuvre

Dans ce paragraphe nous définissons le problème à résoudre et les différentes étapes qui conduiront à la définition des classes et de leurs méthodes.

Orientation

Le travail de mise en œuvre de la méthodologie que nous proposons dans ce rapport s'appuie sur un travail précédent [**?**] qui était une analyse objet de la programmation des éléments finis pour le Fortran-90. Le présent travail est plus ambitieux puisqu'il doit permettre de choisir plusieurs types de méthodes de discrétisation. Les raisons de ce choix sont la rapidité de l'écriture (au moins pour les éléments finis) et la réutilisation d'un savoir faire. Il est clair que les options retenues pour cette mise en œuvre sont propres à ce contexte et pourront être remises en cause dans des versions ultérieures.

4.3.1 Définition du problème

La question posée est la suivante :
(1) On veut résoudre le problème (4.1), pour tout second membre f donné de façon analytique. Le domaine de calcul est le carré unité $[0,1] \times [0,1]$ et on prendra $\Gamma = \partial\Omega$. On cherche les valeurs de la solution u en tous les points d'une grille de pas h donnée.
Cette question se prolonge au niveau discret par :
(2) On cherche à comparer l'efficacité relative des méthodes d'éléments finis et de différences finis.
Enfin au niveau des algorithmes de résolutions de systèmes linéaires :
(3) On voudrait tester différentes méthodes de stokage de la matrice ainsi que les principales méthodes de descente.

4.3.2 Le paquetage `Problem`

Le paquetage `Problem` regroupe les classes `Problem`, `Discret_Problem` et `Algebric_Problem`. Ces classes sont liées par des associations qui sont des agrégations fortes, c'est-à-dire que `Problem` possède `Discret_Problem` qui possède `Algebric_Problem`.

La classe `Problem`

La classe `Problem` (voir 4.1) contient donc toutes les informations nécessaires à la définition du problème et à sa résolution :
- les données physiques du problème : ν et α,
- la définition du domaine de calcul (à travers le paquetage `Domain`) ,
- la liste des points d'observation de la solution (c'est ce qui permettra la remontée de cette solution depuis les étages discrets),
- la liste des opérateurs intervenant dans le problème : le laplacien et l'identité sous la forme d'un opérateur de rigidité (paquetage `Operator`),

– la liste des variables : la solution u et le second membre f (paquetage `Variable`).

Les méthodes suivantes agissent sur ces données :

– Construire le problème à partir de ces données.
– Effectuer le calcul et obtenir la solution. Comme la classe `Problem` possède la classe `Discret_Problem`, cette méthode va pouvoir lancer le calcul au niveau discret, qui lui-même commandera l'inversion des systèmes au niveau algébrique.
– Demander un rapport de comportement.

Il faut aussi envisager des procédures de sauvegarde et de reprise, mais devant la simplicité du problème, elles ne sont pour l'instant pas indispensables.

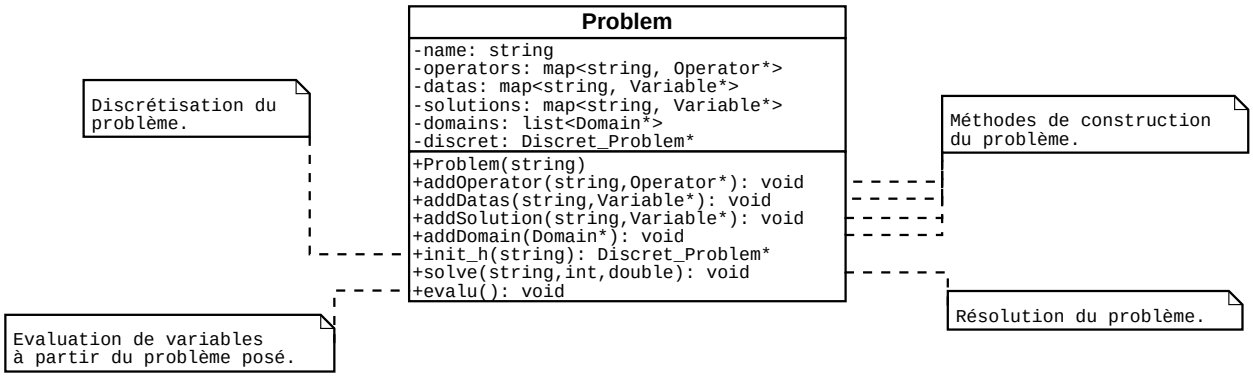

FIG. 4.1 – Classe Problem.

La classe `Discret_Problem`

C'est à ce niveau que le problème à résoudre est discrétisé, c'est-à-dire que les algorithmes de résolution sont définis : intégration en temps, linéarisation des équations et bien sûr discrétisation spatiale.

On y trouve les versions discrètes des entitées définies dans la classe `Problem` :

– la définition de la discrétisation du domaine de calcul,
– la liste des opérateurs discrets,
– la liste des variables discrètes.

On y trouve également les méthodes de construction et de résolution du problème discret qui généreront les entités du niveau suivant (par l'intermédiaire de l'objet `Algebric_Problem`).

Dans cet exemple, l'algorithmique mise en place à ce niveau est très simple puiqu'elle ne consiste qu'en la discrétisation spatiale d'une équation linéaire.

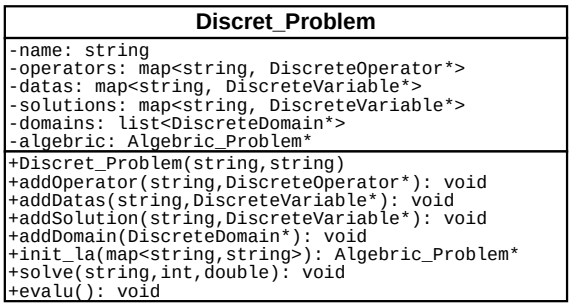

FIG. 4.2 – Classe Discret_Problem.

La classe `Algebric_Problem`

Le dernier niveau du paquetage `Problem` est celui des entités algébriques. On y trouve les matrices et les vecteurs ainsi que les méthodes qui les concernent comme les méthodes de résolution de systèmes linéaires. La construction des matrices dépend de la méthode de stockage (classe `Storage`). Cependant, la manipulation de ces matrices est indépendante de la façon dont elles sont stockées.

La construction des instances de `Algebric_Problem` se fait grâce aux entités discrètes des instances de `Discret_Problem` associées.

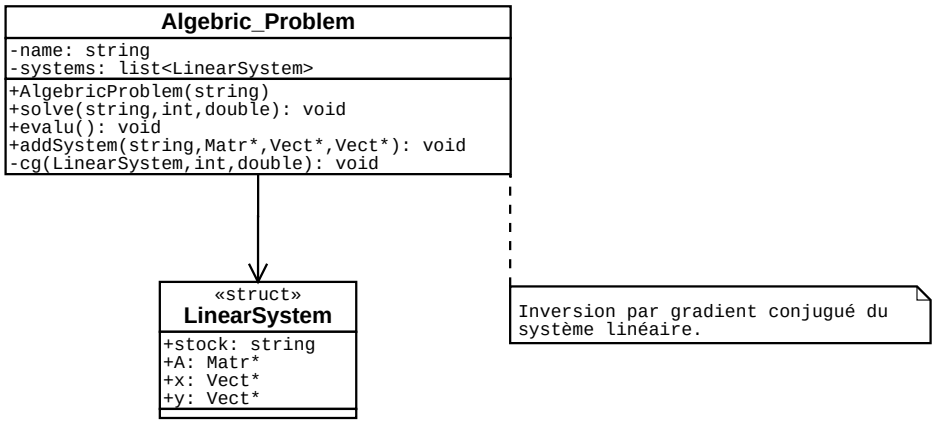

FIG. 4.3 – Classe Algebric_Problem.

4.3.3 Le paquetage `Variable`

Au niveau continu, la classe `Variable` se fait l'expression des espaces fonctionnels utilisés pour la résolution du problème. Au niveau discret, la classe `DiscreteVariable` reflète les espaces discrets associés aux espaces continus du niveau supérieur. Enfin, au niveau algébrique, la classe `Vect` représente les objets informatiques intervenant dans les systèmes linéaires.

4.3.4 Le paquetage `Operateur`

Ce package permet de définir les opérateurs élémentaires nécessaires au problème. Il en assume la discrétisation en fonction de la donnée d'une méthode de discrétisation et conduit à des matrices au niveau algébrique profond par la connaissance d'une méthode de stockage.

Les hiérarchies des opérateurs ont été explicitées en 2. Nous ne détaillons ici que les opérateurs concrets utilisés dans le problème qui nous intéresse. Les opérateurs de masse (`MassOperator`) et de rigidité (`RigiOperator`) sont des spécialisations des classes abstraites `LinearOperator`). Elles sont associées, au niveau discret, aux classes `DiscreteMassOperator` et `DiscreteRigiOperator`. Les instances de ces classes permettent de construire les matrices associées, de façon indépendante de la méthode de stockage choisie.

4.3.5 Le paquetage `Domain`

La classe générique du domaine de calcul doit fournir une description du domaine de calcul mais aussi les informations nécessaires à la remontée des solutions discrètes vers le niveau continu comme l'ensemble des points où l'utilisateur final du programme désire observer les solutions par exemple. Dans le cadre de cette application, tous les domaines sont de dimension 2.

Au niveau continu, un objet `Domain` permet de récupèrer les données indispensables à la construction du domaine discret associé.

Il y a de nombreuses façons de définir le domaine de calcul, soit analytiquement (disque de centre et de rayon donnés, ...), soit discrètement en donnant un maillage initial (définissant les points d'observation des solutions). Ces deux techniques sont possibles par l'intermédiaire des constructeurs.

La classe `DiscreteDomain` est abstraite (4.4). Elle doit accueillir aussi bien des maillages de type éléments finis ou volumes finis que des grilles pour les différences finis.

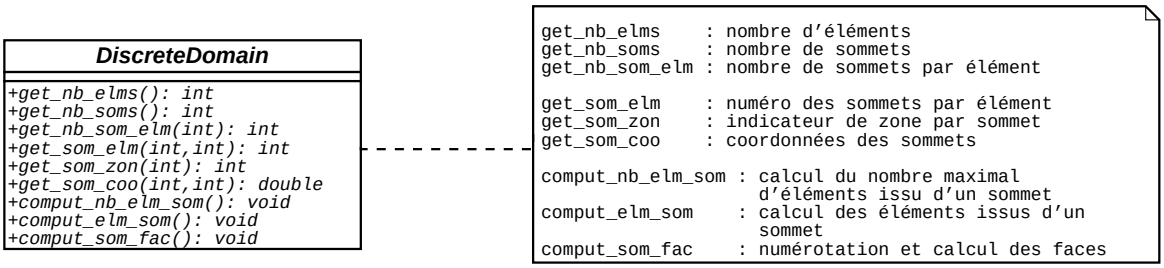
DiscreteDomain
+get_nb_elms(): int
+get_nb_soms(): int
+get_nb_som_elm(int): int
+get_som_elm(int,int): int
+get_som_zon(int): int
+get_som_coo(int,int): double
+comput_nb_elm_som(): void
+comput_elm_som(): void
+comput_som_fac(): void
get_nb_elms : nombre d'éléments
get_nb_soms : nombre de sommets
get_nb_som_elm : nombre de sommets par élément
get_som_elm : numéro des sommets par élément
get_som_zon : indicateur de zone par sommet
get_som_coo : coordonnées des sommets
comput_nb_elm_som : calcul du nombre maximal d'éléments issu d'un sommet
comput_elm_som : calcul des éléments issus d'un sommet
comput_som_fac : numérotation et calcul des faces

FIG. 4.4 – Classe DiscreteDomain.

Le langage utilisé pour nommer les différentes méthodes est celui des éléments finis. Nous proposons deux spécialisations de la classe `DiscreteDomain`, l'une pour les maillages de type éléments finis ou volumes finis (classe `Triangulation`, pour les maillages non structurés), l'autre pour les grilles (classe `Grid`).

Les instances de la classe `DiscreteDomain` disposent de la liste des types de domaine de contrôle qu'elles connaissent. On y définit les conventions de numérotations locales. Cette information est utilisée aussi par le paquetage `Method`.

Maillages non structurés

La notion de maillage est très vague. On entend par maillage dans sa version minimaliste la donnée :

- du nombre d'éléments (ou domaines de contrôle),
- du nombre de sommets,
- des coordonnées des sommets,
- des numéros de sommets par éléments (dans notre exemple tous les éléments sont du même type),
- d'un indicateur par sommet.

A partir de ces informations, nous sommes capables de construire les quantités manquantes comme

- le nombre de cotés (ou faces),
- la liste des extrémités des cotés,
- la liste des éléments partageant une même face,
- la liste des éléments partageant un même sommet,
- des indicateurs par éléments ou par faces.

Une seule méthode permet la construction de ces quantités : `comput_som_fac`. Certains mailleurs permettent la construction de ces quantités directement. Mais afin de respecter les conventions internes des maillages (ordre des sommets, des faces sur un élément par exemple) il est préférable d'entrer le maillage minimal et de construire le reste grâce aux méthodes de la classe `DiscreteDomain`.

Grilles

La notion de grille se caractérise par une numérotation i, j, k des nœuds et des mailles. Les informations comme les numéros des sommets, ceux des éléments (mailles) ne sont pas stockées mais recalculées. Il en est de même pour les coordonnées de sommets. La construction d'une grille demande la connaissance :

– du nombre de points sur les cotés générateurs,
– de la position de ceux-ci qui peut-être donnée explicitement pour les grilles non régulières ou recalculée à partir d'un point d'origine et de pas dans chaque direction pour les grilles régulières.

A partir de ces informations il est possible de construire un maillage et donc de faire des éléments finis sur une grille.

4.3.6 La classe `Method`

La classe générique de la méthode de discrétisation se nomme `Method`. On rappelle que c'est une classe abstraite indépendante du choix de la méthode de discrétisation. Sa finalité est la construction des opérateurs élémentaires restreints à un domaine de contrôle. Ceux-ci sont utilisés par les instances des classes concrètes dérivées de la classe `Operator` pour la discrétisation des opérateurs du problème (4.1).

Cette classe utilise le maillage du domaine de calcul (classe `DiscreteDomain`). Elle est capable de fournir par opérateur (voir 4.5 et 4.6) :

– les nombres de lignes et de colonnes globaux ,
– le nombre de domaine de contrôle,
– par domaine de contrôle, le nombre de lignes et de colonnes,
– les matrices (ou les coefficients de la matrice) de la représentation discrète de l'opérateur élémentaire.

L'exemple d'application nous conduit à spécialiser la classe `Method` pour les opérateurs élémentaires du problème traités, c'est à dire l'opérateur de masse et l'opérateur de rigidité.

Ces méthodes sont complètement spécifiées dans la documentation interne. On utilisera une classe de type `factory` pour la sélection de la méthode de discrétisation. Les sous-paragraphes suivants donnent l'implémentation des méthodes virtuelles pures précédentes pour chacune des techniques de discrétisation considérées ici.

4.3.7 Eléments finis

La mise en œuvre proposée ici est très proche de celle utilisée dans [**?**] qui est une analyse objet de la programmation pour des problèmes discrétisés par éléments finis adaptés au langage Fortran-90. On utilise des méthodes identiques pour la construction des fonctions de bases et des méthodes d'intégration numérique. Les éléments finis P0, P1, P1 non conforme, P2, Q0, Q1 et Q2 sont disponibles (en dimension 2). On dispose des formules d'intégration de Gauss de degré 1, 2, 3 ainsi que de formules dont les points d'intégration sont les degrés de liberté des éléments finis. Ces dernières formules d'intégration sont bien entendu moins précises que les formules de Gauss, mais elles permettent de nombreuses vérifications (par exemple les sommets pour le Laplacien donne le même stencil que les différences finis d'ordre 2 (-1, -1, 4, -1, -1)).

Structure de la classe `Finite_Elements_Pk`

La classe `Finite_Elements_Pk` a été conçue pour pouvoir utiliser tous les éléments finis de Lagrange, quelque soit la dimension d'espace. On définit d'abord la méthode d'intégration numérique et pour chaque famille d'intégration on lui rattache la définition des éléments finis, assurant ainsi leur compatibilité.

La définition d'une méthode d'éléments finis comprend deux sous-ensembles, un pour préciser les degrés de liberté et l'autre pour les fonctions de bases et leurs dérivées. Tout est donné sur un élément de référence. Les fonctions de bases sont connues par l'intermédiaire de leurs valeurs aux points d'intégration (ainsi que la valeur de leurs dérivées). On a ainsi toute l'information nécessaire à la construction de la liste des degrés de liberté. Par exemple, si on demande les éléments P2, les données suivantes sont requises :

1. il y a des degrés de liberté sur les cotés, si on ne dispose pas d'une numérotation de ceux-ci, on demande à l'objet `DiscreteDomain` de la construire.

2. On connait donc le nombre des degrés de liberté et leurs numéros par éléments.
3. On peut également avoir besoin de la liste des nœuds voisins (définissant le graphe de la matrice).

Ainsi, connaissant les valeurs des fonctions de base aux points d'intégration on est capable de calculer l'ensemble des opérateurs aux dérivées partielles.

Chaque instance de la classe `Method` connaît l'objet de maillage sur lequel est résolu le problème. Outre la numérotation des degrés de liberté, on construit l'application de l'élément de référence à l'élément courant.

Le diagramme UML des différentes classes de la hiérarchie `Method` relatives aux éléments finis ainsi que les classes utilisées par cette hiérachie est présenté en 4.5.

Comment ajouter un domaine de référence

Les domaines de contrôle de référence (`Ref_Elmt`) sont identifiés par une chaine de caractères (qui est donnée au constructeur). On dispose des domaines suivants :

en dimension 1

`SEGM-1D` un segment

en dimension 2

`TRIA-2D` un triangle

`RECT-2D` un carré

en dimension 3

`TETR-3D` un tétraèdre

`CUBE-3D` un cube

`PRIS-3D` un prisme à base triangulaire

Pour chaque type on précise la dimension d'espace, le nombre de sommets et leurs coordonnées. Des méthodes publiques donnent accés à ces informations : `dims`, `nb_soms` et `som_coo` respectivement.

Pour ajouter un domaine de référence, il suffit de modifier le constructeur de la classe `Ref_Elmt` en insérant ce nouveau type.

Comment ajouter une méthode d'intégration

Une méthode d'intégration numérique est construite à l'aide d'un domaine de référence et d'une chaîne de caractères spécifiant le type d'intégration souhaité. Les méthodes de la classe `Num_Intg` permettent de connaître le nombre de points d'intégration `numb`, les coordonnées de ceux-ci `coor` et les coefficients d'intégration `coef`.

Les types d'intégration actuellement disponibles sont :

sur un segment

sur un triangle

`2D-TRIA-S-3` Les points d'intégrations sont les sommets

`2D-TRIA-G-1` Un point de Gauss

`2D-TRIA-G-3` Trois points de Gauss

`2D-TRIA-G-4` Quatre points de Gauss

sur un carré

`2D-QUAD-S-4` Les points d'intégrations sont les sommets

`2D-QUAD-G-4` Quatre points de Gauss

L'ajout d'un nouveau type d'intégration se fait simplement en modifiant le constructeur de la classe par l'insertion de ce nouveau cas.

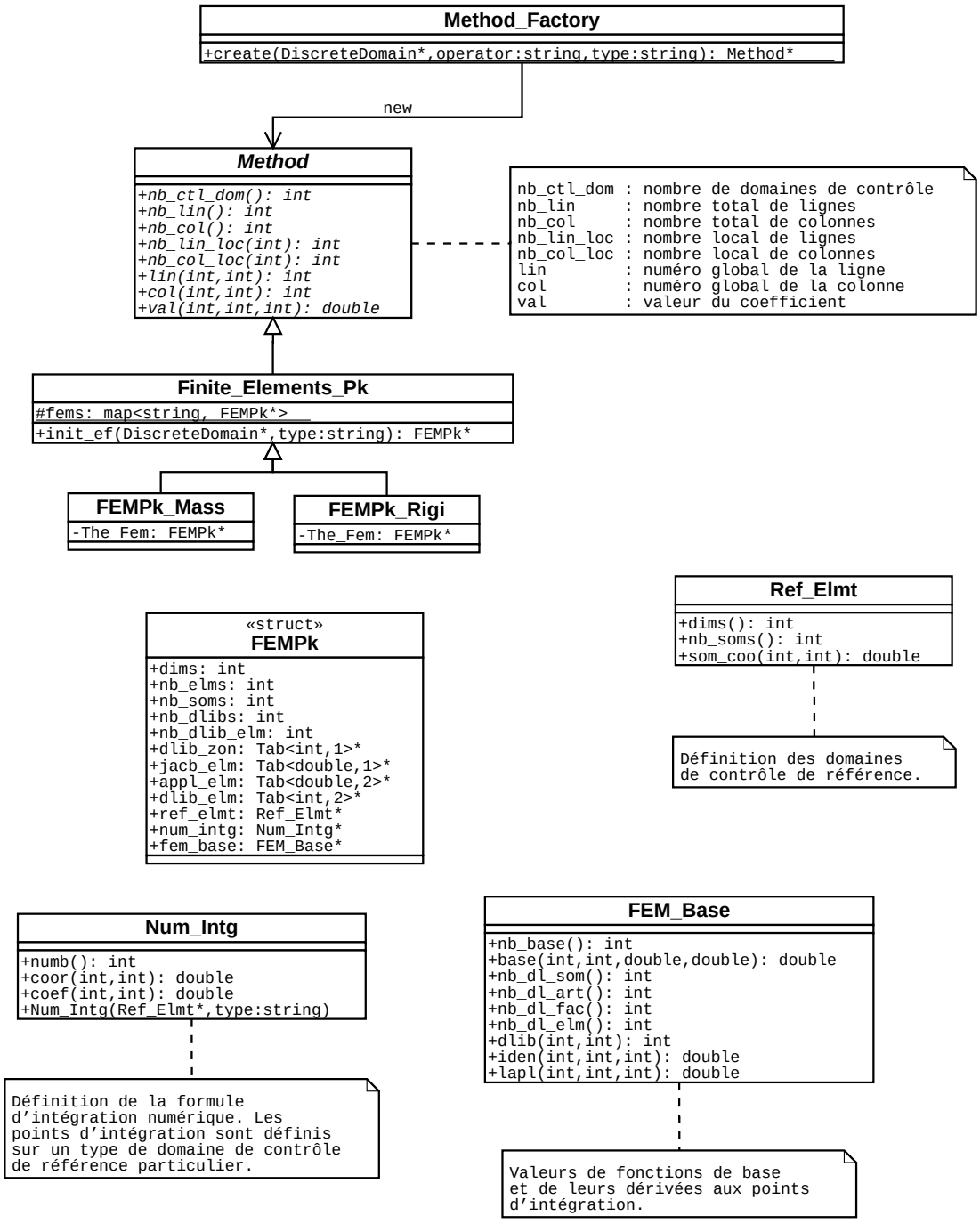

FIG. 4.5 – Hiérarchie de la classe `Method` relative aux éléments finis.

Comment ajouter un type d'éléments finis

Les éléments finis actuellement disponibles sont :

sur un triangle

`P0` élément de degré 0 par élément

`P1` élément de degré 1 par élément

`P1NC` élément non conforme de degré 1 par élément

`P1BUL` élément de degré 1 par élément avec bulle

`P2` élément de degré 2 par élément

sur un carré

`Q1` de degré 1 dans chaque direction par élément

L'ajout d'un nouveau type d'élément fini se fait par spécialisation de la classe générique `Method`, sur le modèle de ce qui existe pour les éléments finis Pk.

4.3.8 Différences finis

On propose des différences finis d'ordre deux et d'ordre quatre. Elles sont basées sur des définition en dimension 1, les dimensions supérieurs sont obtenues par produit extérieur. L'implémentation proposée donne des formules différentes pour les points proches du bord du même ordre que celle des points intérieurs.

Le diagramme UML des différentes classes de la hiérarchie `Method` relatives aux différences finies ainsi que les classes utilisées par cette hiérachie est présenté en 4.6.

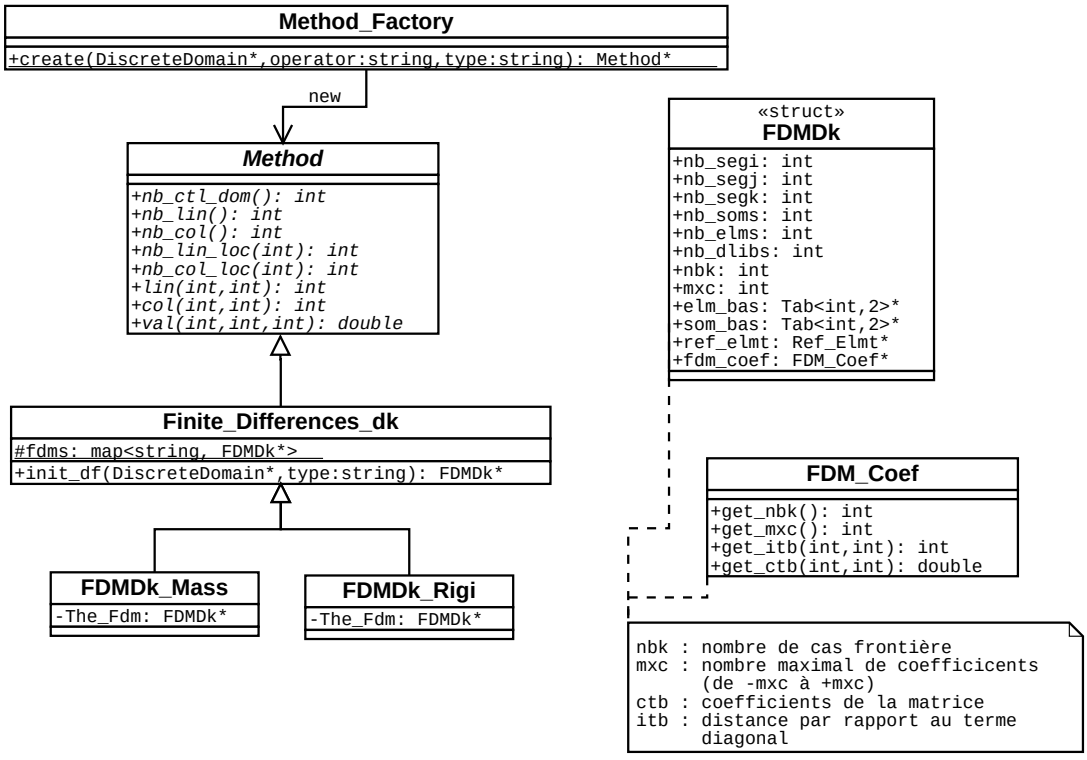

FIG. 4.6 – Hiérarchie de la classe `Method` relative aux différences finies.

4.3.9 La classe `Storage`

On propose trois types de stockage des matrices, celui des vecteurs est trivial et est représenté par des tableaux monodimentionnels. Pour chaque méthode de stockage on doit définir les méthodes d'accés aux lignes (ou aux colonnes) des matrices, de produit matrice-vecteur et de remplissage (4.7).

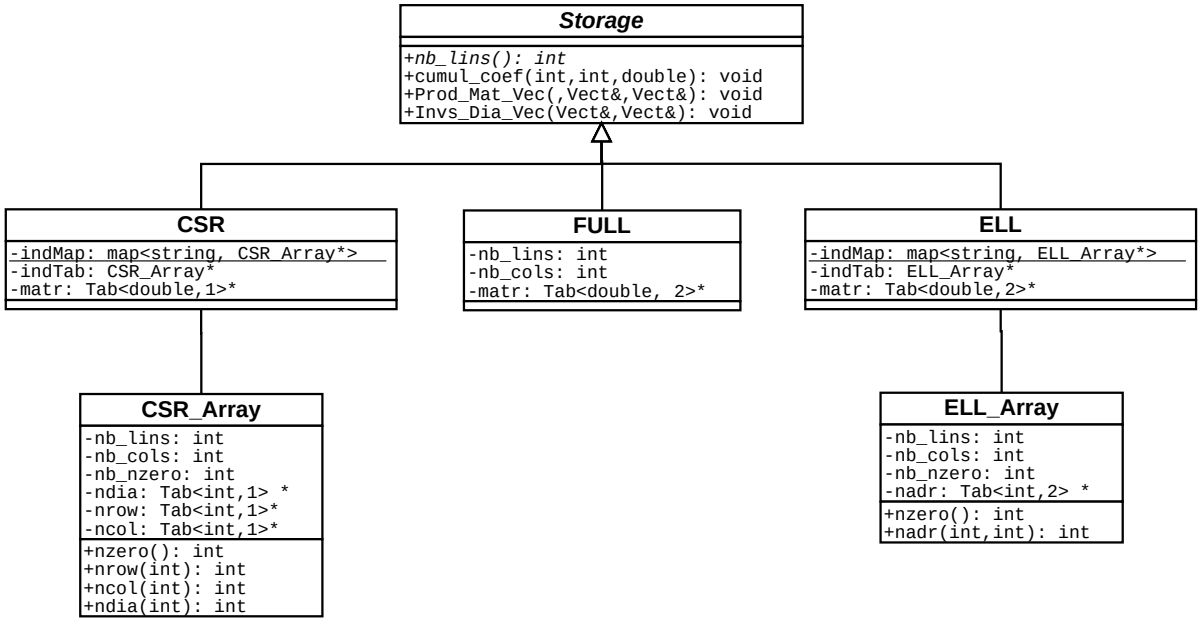

FIG. 4.7 – Hiérarchie de la classe `Storage`.

Le format ELL

Le format ELL (voir bibliotheque ELLPACK) utilise des tableaux à deux indices pour stocker la matrice. Il est nécessaire de connaître le nombre maximal de coefficients non nuls par ligne. Ce stockage n'est pas optimal de ce fait. Le produit matrice-vecteur s'écrit, en utilisant les noms des attributs des classes concernées (`ELL` et `ELL_Array`) :

```
for (i = 1; i++ ; i <= nb_lins+1)
  {
   y(i) = 0.0;
   for (k = 1; k++ ; k < nb_coef_lin)
     {
      j = nadr(i,k);
      y(i) = y(i) + matr(i,k) * x(j);
     }
  }
```

Le format CSR

La méthode de stockage CSR (Compress Sparse Row) est une méthode plus compacte que la précédente (ELL). On stocke à la suite les uns des autres (dans des tableaux mono-indicés) et ligne par ligne les coefficients non nuls et les numéros de colonnes correspondantes. A ces deux informations on ajoute un pointeur sur chaque début de ligne. Le produit matrice-vecteur s'écrit, en utilisant les noms des attributs des classes concernées (`CSR` et `CSR_Array`) :

```
for (i = 1; i++ ; i <= nb_lins+1)
  {
   y(i) = 0.0;
   for (k = nrow(i); k++ ; k < nrow(i+1))
     {
      j = ncol(k);
      y(i) = y(i) + matr(k) * x(j);
     }
  }
```

4.3.10 Construction informatique du problème elliptique

Nous venons de décrire les paquetages constituants la librairie. Nous allons maintenant présenter les instructions informatiques permettant la résolution du problème elliptique que nous avons choisi comme exemple d'application.

Le programme principal se déroule en plusieurs étapes :

1. Construction de la géométrie du problème à résoudre,
2. Construction des opérateurs du problème,
3. Construction des variables données et solution intervenant dans les équations,
4. Choix de la méthode de discrétisation et discrétisation du problème continu,
5. Choix de la méthode de stockage par opérateur et linéarisation du problème discret,
6. Résolution du problème par inversion du problème algébrique correspondant,
7. Visualisation de la solution.

Construction de la géométrie

Considérons par exemple le choix d'une triangulation constituée de mailles rectangulaires. On construit le domaine continu et on l'ajoute au problème. On obtient ainsi les instructions suivantes :

```
// Pb construction
Problem problem("Test");

// Continuous Domain
int nbseg1, nbseg2, opt_dom_elm, opt_dom_som;

cout << "TST-DOMAIN : enter nbseg1 ? "; cin >> nbseg1; cout << endl;
cout << "TST-DOMAIN : enter nbseg2 ? "; cin >> nbseg2; cout << endl;

opt_dom_som = 0;
opt_dom_elm = 0;

Domain *geom =  new Domain("TRIA", "RECT-2D",
            nbseg1     , nbseg2,
    opt_dom_elm, opt_dom_som);

problem.addDomain(geom);
```

Construction des opérateurs et des variables

Dans l'exemple traité, nous avons besoin de l'opérateur de rigidité, associé au second membre et à la solution. Les instructions suivantes permettent la construction de ces entités et leur ajout au problème traité.

```
// Continuous operators
RigiOperator A(geom); // Linear Operator of the test problem

// Continuous Variable
VolumeVariable ADatas(geom);  // For Linear Operator
VolumeVariable   ASol(geom);  // For Linear Operator

problem.addOperator("Rigi", &A);
problem.addDatas   ("Rigi", &ADatas);
problem.addSolution("Rigi", &ASol);
```

Discrétisation du problème continu

La méthode de discrétisation est choisie interactivement par l'utilisateur, et permet la discrétisation du problème continu par l'intermédiaire de la méthode `init_h`.

```
// Caracterization of the discretisation method
string meth_id;

cout << "TST-PROBLEM : Enter the discretization Method " <<endl;
cout << "              ATTENTION for FEM you must add INTG (FEM%INTG) ? ";
cin  >> meth_id;
cout << endl;

// Discretization of the problem
DiscreteProblem* problem_h = problem.init_h(meth_id);
```

Linéarisation du problème discret

Le choix de la méthode de stockage se fait pour chacun des opérateurs du problème, de façon interactive. Elle permet de pouvoir générer le problème algébrique associé au problème discret par le biais de la méthode `init_la`

```
// Caracterization of the storage
map<string, string> stock;
string stor_id;

cout << "TST-PROBLEM : Enter the Storage Method " <<endl;
cin  >> stor_id;
cout << endl;

stock["Rigi"] = stor_id;

// Construction of the algebric problem
problem_h->init_la(stock);
```

Résolution du problème et visualisation de la solution

La résolution du problème au niveau continu par la méthode `solve` entraîne en cascade l'inversion des systèmes algébriques. La visualisation de la solution, sous format texte, se fait simplement en utilisant l'opérateur `«`.

```
// Solving the problem

// Second member initialization
```

```
ADatas = 1.0;

// Solution initialization
ASol = 0.0;

// Resolution with cg

int    itmax   = 100;
double epsilon = 0.000001;

problem.solve("CG", itmax, epsilon);

cout << "Solution Rigi Operator : " << endl;
cout << ASol << endl;

problem.evalu();

cout << "Verification (must be one) " << endl;
cout << ADatas << endl;

// View the solution
//   cout << "Solution Mass Operator : " << endl;
//   cout << MSol << endl;
```

Chapitre 5

Pratique de l'environnement de travail

Introduction

Cette partie correspond à un guide d'utilisation de l'environnement de programmation et d'exploitation présenté précédemment.

L'instrumentation d'un code de calcul se fait par l'intermédiaire d'une bibliothèque de classes disponibles en différents langages et qui permettent de partager les données du codes via le serveur CORBA.

La première section correspond à la phase de démarrage du serveur de données.

La deuxième section présente la démarche d'instrumentation d'un code de calcul en C++, en se basant sur l'exemple concret de l'instrumentation des sources de la partie mathématique du logiciel GTOOCS.

Enfin, la dernière section détaille l'utilisation de l'interface graphique permettant l'exploitation des codes instrumentés.

L'ensemble des exécutables et des librairies utilisés dans cette partie a été décrit et localisé physiquement dans la partie précédente par un diagramme de composants.

5.1 Démarrage du serveur

La mise en route du serveur doit se faire en deux temps : il faut d'abord démarrer les services CORBA nécessaires au bon fonctionnement du logiciel, puis mettre le serveur en attente des requêtes des clients.

5.1.1 Services CORBA

Les services CORBA indispensables au lancement du serveur sont :
- le service de nommage,
- le service d'évènement.

Dans le cas de l'utilisation de l'implémentation CORBA d'Orbacus, le démarrage de ces services se fait de la façon suivante :

```
nameserv  -OAport 1977 -OAhost localhost &
eventserv -OAport 1978 -OAhost localhost &
```

L'option *-OAport* désigne le port de la machine sur lequel est accessible le service, l'option *-OAhost* désigne la machine sur lequel le service est démarré. Ainsi, dans l'exemple précédent, le service de nommage est démarré sur la machine locale, en attente sur le port 1977, et le service d'évènement est démarré sur la machine locale, en attente sur le port 1978.

5.1.2 Serveur

Le démarrage du serveur se fait en lançant l'exécutable, idéalement en batch avec récupération des sorties dans un fichier de log. L'exécution du serveur se fait sous la forme suivante :

```
./server -ORBInitRef NameService=corbaloc:iiop:localhost:1977/NameService
-ORBInitRef EventChannelFactory=corbaloc:iiop:localhost:1978/DefaultEventChannelFactory
```

L'option -ORBInitRef permet d'indiquer les références initiales :
- du service de nom qui est en attente sur la machine locale et le port 1977,
- de la fabrique des canaux d'évènements qui est en attente sur la machine locale et le port 1978.

5.2 Instrumentation d'un code

Pour décrire de façon pratique l'instrumentation d'un code de calcul, nous allons nous baser sur un code C++ réalisé à partir des classes de la bibilothèque mathématique et numérique du logiciel GTOOCS. La mise en oeuvre pour un code implémenté dans un autre langage se fait de manière analogue.

5.2.1 Langages disponibles

La librairie des classes d'instrumentation de codes utilise les interfaces IDL des objets communiquants. Elle est donc disponible pour les langages générés par les compilateurs idl c'est-à-dire essentiellement :
- C++,
- Java.

Les codes de calculs étant souvent développés en Fortran, la bibliothèque des classes d'instrumentation est aussi disponible sous forme encapsulé en :
- Fortran 77,
- Fortran 90.

Le choix de ces langages a été fait pour trois raisons principales :
- La prise en compte de l'existant, constitué essentiellement de codes en fortran (77 ou 90),
- La volonté de développer les nouveaux codes à l'aide d'une conception orientée objet incitant plutôt à l'utilisation des langages objets comme le C++,
- la nécessité de proposer une interface graphique compatible avec le serveur CORBA, naturellement développée en Java pour des raisons essentielles de portage.

5.2.2 Compilation

L'instrumentation d'un code utilise des biblitohèques qu'il faut inclure dans la compilation. Ainsi, la génération des fichiers objets correspondants aux fichiers instrumentés nécessite l'inclusion, sur la ligne de compilation, des fichiers entêtes contenus dans les répertoires suivants :
- `Client/CPP` (classes instrumentales),
- `SQL/CPP` (classes utilisateurs),
- `IDL/CPP` (classes des objets distribués).

De façon analogue, la génération de lien pour la création d'un exécutable nécessite la prise en compte des librairies suivantes :
- `Client/CPP/client_cpp.a,`
- `SQL/CPP/sql_cpp.a,`
- `IDL/CPP/corba_cpp.a.`

D'autre part, il faut également faire le lien avec les bibliothèques de l'ORB Orbacus sous la forme :
`-lOB -lJTC -lpthread -lCosNaming -lCosEvent`

5.2.3 Initialisations, finalisation

Le programme principal du code instrumenté doit faire appel à une méthode d'initialisation, *init(...)*, déclarée dans l'entête *Global.h*, et dont le rôle est multiple :

- initialiser les références CORBA,
- instancier l'objet unique Process,
- instancier les objets miroirs des fabriques d'objets distribués.

L'appel de cette fonction doit idéalement être la première instruction du programme.

De façon analogue, le programme principal doit se terminer par une méthode de finalisation, *end()*, également déclarée dans l'entête *Global.h*, et qui contrôle entre autre les sauvegardes et/ou destructions des données sur le serveur.

```
#include "Global.h"

int main( int argc, char{*}{*} argv)
{
   init(argc, argv, ‘‘TST-OPERATOR", PERSISTENT, 7, 4, 4);

   ...

   end();

   return 0;
}
```

La fonction `end()` ne comporte pas d'argument.

La fonction `init(...)` prend au minimum trois arguments, au maximum sept arguments :

- les deux premiers arguments sont ceux passés dans l'appel de la routine principale `main`.
- Le troisième argument, de type `string`, correspond au nom de l'exécution, qui apparaîtra lors de l'exploitation graphique. Ce nom doit être unique sur le serveur. Si le nom choisit existe déjà, le programme rajoute un `‘‘0’’` au nom (jusqu'à obtenir un nom non existant sur le serveur), et continue l'exécution.
- Le quatrième argument, optionel, de type `int`, correspond au temps de vie, sur le serveur, de l'exécution et des données qui lui sont associées. Cette argument peut prendre deux valeurs :

1. TRANSIENT : l'exécution et les données sont détruites du serveur à la fin du process. C'est la valeur par défaut.
2. PERSISTENT : l'exécution et les données sont conservées sur le serveur, et restent accessibles après la fin du process.

- Les arguments 5, 6 et 7, de type `int`, sont les permissions associées à cette exécution, correspondantes respectivement au propriétaire, au groupe du propriétaire et au reste du monde, c'est-à-dire de la forme des permissions UNIX, avec des valeurs de même signification :

1. permission de 4 : permission de lecture, qui autorise l'accès à l'exécution et aux données associées, mais pas au contrôle de l'exécution,
2. permission de 2 : permission d'écriture, permettant la modification des données (non implémenté dans cette version),
3. permission de 1 : permission d'exécution, autorisant le contrôle de l'exécution.

Dans le cadre de l'exemple, l'exécution sera accessible en lecture par tous (propriétaire, groupe et reste du monde) et en écriture et exécution par le propriétaire uniquement, qui sera seul autorisé à contrôler cette exécution.

C'est également le comportement par défaut.

5.2.4 Sortie standart : Cout

La bibliothèque d'instrumentation offre la possibilité de distribuer la sortie standart par l'utilisation de la variable globale `Cout`, de type `StdCout`, et déclaré dans le fichier d'entête `Global.h`.

Il est donc possible de remplacer, là ou on le souhaite, la variable standart `cout` par cette nouvelle variable `Cout`, qui, à chaque appel de l'opérateur `«`, réalise deux actions :

– écriture dans un objet distribué,
– écriture sur la sortie standart.

Ainsi, l'action locale de cette variable est transparente par rapport à l'utilisation de la variable classique *cout*.

La modification du programme principal de notre exemple est illustré dans l'extrait suivant des sources :

```
#include "Domain.h''
#include "Finite_Elements_Pk.h''
#include "Global.h''

int main( int argc, char{*}{*} argv)
{

  init(argc, argv, "TST-OPERATOR", PERSISTENT, 7, 4, 4);

  string option;

  Cout << " TST-OPERATOR : Read a domain " << endl;

  Domain geom("TRIA", "../Domain/test-2d.mesh", "MESH");

  Cout << geom << endl;

  Cout << " TST-OPERATOR : enter an OPTION ? ";

  while(cin >> option)
     {

        Cout << endl;

        if      (option == "EXIT"})
           break;
        else if (option == ``TEST")
           {

              DiscreteDomain {*}dgeom = geom.get_ddom();

              string fems_id = "P1";
              string intg_id = "2D-TRIA-G-3";

              Finite_Elements_Pk FEMP1(dgeom, intg_id, fems_id);

              double Dt = 1.0;
              double Re = 1.0;

              Cout << `` TST-OPERATOR : Dt = `` << Dt << endl;
              Cout << `` Re = `` << Re << endl;
```

```
...
```

Il faut noter que l'utilisation de cette variable pour des objets pour lesquels l'opérateur « a été redéfini, nécessite de la même manière la redéfinition de cette opérateur pour la classe StdCout comme le détaille l'exemple suivant, concernant la hiérarchie des classes de stockage :

Fichier Storage.h :

```
#include "Cout.h"

class Storage
   {

      ...

      // Surcharge de l'operateur de redirection <<

      friend ostream& operator << (ostream&,const Storage&);

      // Surcharge de l'operateur de redirection <<

      friend StdCout& operator << (StdCout&, const Storage&);

      // Méthode d'impression à implémenter dans les classes dérivées

      virtual ostream& print(ostream&) const = 0;

      // Méthode d'impression à implémenter dans les classes dérivées

      virtual StdCout& print(StdCout&) const = 0;

      ...

   }
```

Fichier Storage.cpp :

```
// ------------------------------------------------------------------
// Surcharge de l'operateur de redirection <<
ostream& operator << (ostream& os, const Storage& sto)
{
  return sto.print(os);
}


// ------------------------------------------------------------------
// Surcharge de l'operateur de redirection <<
StdCout& operator << (StdCout& os, const Storage& sto)
{
  return sto.print(os);
}
```

Fichier FULL.h :

```
class FULL : public Storage

{

 ...

 protected:

  // Print Method
  ostream& print(ostream&) const;
  StdCout& print(StdCout&) const;

 ...

};
```

Fichier FULL.cpp :

```
...

// ---------------------------------------------------------------------
// Print Method
ostream& FULL::print(ostream& os) const
{
  os << "...Print FULL : " << endl;
  int i,j;
  for (i = 1 ; i <= _nb_lins ; i++) {
    for (j = 1 ; j <= _nb_cols ; j++)
      os << j << "  " ;
    os << endl;
    os << i << "  " ;
    for (j = 1 ; j <= _nb_cols ; j++)
      os << (*_matr)(i,j) << "  ";
    os << endl;
  }
  return os;
}

// ---------------------------------------------------------------------
// Print Method
StdCout& FULL::print(StdCout& os) const
{
  os << "...Print FULL : " << endl;
  int i,j;
  for (i = 1 ; i <= _nb_lins ; i++) {
    for (j = 1 ; j <= _nb_cols ; j++)
      os << j << "  " ;
    os << endl;
    os << i << "  " ;
    for (j = 1 ; j <= _nb_cols ; j++)
      os << (*_matr)(i,j) << "  ";
    os << endl;
  }
  return os;
```

```
}

...
```

5.2.5 Profiling du code : Flow trace

Cet outil, partie intégrante de l'environnement de programmation, permet aux utilisateurs d'analyser le coût en temps CPU et d'établir un profil de fonctionnement d'un programme de calculs.

Son utilisation est optionnelle et particulièrement simple. Deux fonctions globales, déclarées dans la fichier d'entête *Global.h*, permettent de contrôler le processus d'analyse :

- *push(string)* : Cette fonction désigne le début d'une zone d'analyse. Une zone d'analyse est en général constituée des instructions d'une méthode ou d'une fonction et il est bien sur possible d'emboiter plusieurs zones d'analyse. L'argument de type string permet d'identifier chaque zone. C'est, par exemple, de façon classique, le nom de la fonction ou de la méthode analysée.
- *pop()* : Cette fonction signale la fin d'une zone d'analyse. Cet appel correspond au dernier appel push, l'imbrication des zones se faisant sur le principe d'une pile FILO.

L'initialisation du flow-trace se fait lors de l'instanciation de l'objet *Process*. L'analyse finale du flow-trace se fait lors de l'appel de la fonction *end()* détaillée en 5.2.3.

L'exemple suivant détaille l'utilisation du flow-trace dans le programme principal de notre code de test.

```
#include "Global.h"

int main( int argc, char** argv) }*

{
    init(argc, argv, "TST-OPERATOR",PERSISTENT, 7, 4, 4);

    string option;

    push("Main");

    Cout << "TST-OPERATOR : Read a domain " << endl;

    Domain geom("TRIA", "../Domain/test-2d.mesh","MESH");

    Cout << geom << endl;

    Cout << "TST-OPERATOR : enter an OPTION ? ";

    while(cin >> option) {

    ...

    }

    Cout << endl;
    pop();
    end();
```

```
    return 0;

}
```

Enfin, ce dernier exemple expose la mise en oeuvre du flow-trace dans une méthode d'une des classes du projet, issue du fichier *DiscreteProblem.cpp*.

```
// ----------------------------------------------------------------------
// Solving the problem
void DiscreteProblem::solve(string inv, int itmax, double eps)
{
  push(``DiscreteProblem::solve'');
  if (_ind_la == 1) _algebric->solve(inv, itmax, eps);
  pop();
}
```

5.2.6 Instrumentation d'un fichier source

L'instrumentation d'une zone de programme se fait par l'intermédiaire d'une l'instance de la classe *Context*. Cette instance permet :
- de créer des flux distribués,
- d'accéder simplement à ces flux distribués.

Création des flux distribués

La création des objets flux distribués se fait par l'intermédiaire des méthodes *create* de la classe *Context* :
- *create(string nom)* : cette méthode permet la création d'un flux de données non formatté dont le nom est donné en argument.
- *create(string nom, string type, int nbre)* : cette méthode permet la création d'un flux de données formatté nommé *nom* et constitué d'un ensemble de *"nbre"* valeur de type *"type"*.
- *create(string nom, int nbre, string type1, int nbre1, string type2, int nbre2 ...)* : cette méthode permet la création d'un flux de données formatté nommé *nom* et constitué de *nbre* éléments : le premier est un ensemble de *nbre1* valeur de type *type1*, le deuxième est un ensemble de *nbre2* valeur de type *type2* ...

Les noms de flux doivent être uniques pour une exécution donnée. Comme pour le nom de l'exécution, si ce nom existe déjà, le programme ajoute un *"0"* jusqu'à obtenir un nom non utilisé.

Les types disponibles sont :
- *"string"* : type chaîne de caractère,
- *"int"* : type entier,
- *"double"* : type réel double précision,
- *"float"* : type réel simple précision.

L'exemple ci-dessous illustre la création des flux distribués.

Fichier AlgebricProblem.h :

```
#include <string>

#include "Matr.h"
#include "Vect.h"
#include "Context.h"
```

```
class AlgebricProblem{

{
 private:

 ...

   Context ctxt;

  // Solve a linear system with Conjugate Gradient
  void cg (LinearSystem, int, double);

 ...

};
```

Fichier AlgebricProblem.cpp

```
#include "AlgebricProblem.h"
#include "Global.h"

// ------------------------------------------------------------------------
// Constructor
AlgebricProblem::AlgebricProblem(string nm) : _name(nm) {

  ctxt.create("Error", 2,"int", 1,"double", 1);

  ctxt.create("Solution",2,"int", 1,"double", 1);

  ctxt.create("Residu", 2,"int", 1, "double", 1);
}
```

Dans cet exemple, trois flux distribués formattés sont créés. Ils sont tous constitués d'un entier et d'un réel double précision et vont permettre de visualiser les données d'erreur, et de normes des solutions, résidus... lors de la résolution du problème.

Accession et manipulation des flux distribués

L'accession des flux distribués se fait par l'intermédiaire de l'objet *Context*, grâce aux opérateurs *()* et *[]* :

- *opérateur ()(string nom)* : permet l'accès au flux non formatté nommé *nom*.
- *opérateur [](string nom)* : permet l'accès au flux formatté nommé *nom*.

La manipulation de flux se fait, comme pour la bibliothèque standard, par l'intermédiaire de l'opérateur `«`.

L'exemple suivant, faisant suite à l'exemple précédent, illustre le remplissage des données des flux distribués.

Fichier AlgebricProblem.cpp :

```
// ------------------------------------------------------------------------
// Iterative  inversion of a linear system using Conjugate Gradient
void AlgebricProblem::cg(LinearSystem sys, int mxit, double epsl) {

  // sys.A = matrice
```

```
  // sys.y = solu
  // sys.x = smbr

  Matr& A    = *sys.A;
  Vect& solu = *sys.y;
  Vect& smbr = *sys.x;

  Cout << "CG algorithm ...." << endl;
  Cout << "   nb. max. it : " << mxit << endl;
  Cout << "   epsilon     : " << epsl << endl;

  ...

  for (itr = 1; itr <= mxit; itr++)
    {

      A.Prod_Mat_Vec(desc, work);  // work = A * desc;

      aa1 = dot_prod(desc, work);

      if (aa1 < 1.0e-15)
{
  Cout << "PCG ERROR aa1  too small ..." << endl;
  break;
}

      a1 = aa /aa1;

      solu.idpay( a1, desc);  // solu = solu + a1 * desc
      resi.idpay(-a1, work);  // resi = resi - a1 * work

      nsolu = dot_prod(solu, solu);
      ndesc = dot_prod(desc, desc);
      nresi = dot_prod(resi, resi);

      error = a1*a1 * ndesc;

      Cout << "CG " << itr
   << " "  << error
   << " "  << nsolu
   << " "  << ndesc
   << " "  << nresi
   << " "  << aa
   << " "  << aa1
   << " "  << a1
   << " "  << bb1 << endl;

      ctxt[``Error''] << it << error;
      ctxt[``Solution] << it << nsolu;
      ctxt[``Residu] << it << nresi;


      if (error < epsl*epsl) break;
```

```
    ...
}
```

5.3 Exploitation graphique

Cette interface graphique est proposée pour simplifier la visualisation et l'analyse des flux distribués disponibles sur le serveur. Elle a été développée en Java :

- pour des raisons de portabilité,
- pour bénéficier de la génération directe de code par les compilateurs idl.

Elle est constituée d'une fenêtre principale dans laquelle s'ouvrent d'autres fenêtres secondaires.

Les exemples graphiques présentés dans les sections suivantes sont issus de l'instrumentation précédente et concerne un code utilisant les librairires mathématiques du projet GTOOCS.

5.3.1 Démarrage de l'interface

L'interface graphique utilise les classes Java situées dans les répertoires suivant :

- *SQL/JAVA* (classes utilisateurs),
- *IDL/JAVA* (classes des objets distribués)
- *CLIENT/JAVA* (classes instrumentales).

D'autre part, l'utilisation de l'implémention Corba Orbacus, ainsi que les bibliothèques de traçage de courbes jfreechart nécessite de rajouter dans la variable d'environnement CLASSPATH la ligne suivante :

/usr/local/src/jfreechart-0.9.4/lib/jcommon-0.7.1.jar :/usr/local/src/jfreechart-0.9.4/lib/jfreechart-0.9.4.jar :/usr/local/lib/OBEvent.jar :/usr/local/lib/OB.jar

Le main contrôlant le démarrage de l'interface graphique se trouve dans le fichier *ViewManagement.java*. Le fichier de configuration contenant les références initiales des objets Corba est *ORB.config*.

L'exécution est lancé par la ligne de commande (avec une variable CLASSPATH initialisée) :

javaViewManagement -ORBconfig ORB.config

5.3.2 Fenêtre principale d'accueil

Lors du lancement de l'exécutable de l'interface graphique s'ouvre la fenêtre principale d'accueil. Elle comprend une barre de menu, constitué de trois menus :

- *Session*, constitué des items *Exit* (sortie du logiciel) et *Connexion au serveur* (permet la connexion au serveur),
- *Windows*, qui sera constitués des items portant les noms de toutes les fenêtres ouvertes, et éventuellement iconifiées.
- *Update*, constitué de l'item *Update All Now* (mise à jour immédiate des fenêtres ouvertes par rapport aux données du serveur) et de l'item *Update Frequency* (permet de définir la fréquence de mise à jour des fenêtres par rapport aux données du serveur). Par défaut, la fréquence de mise à jour est de 10 secondes.

5.3.3 Fenêtre de connexion

Lors de l'exécution de l'interface graphique, la fenêtre de connexion s'ouvre immédiatement à la suite de la fenêtre d'accueil. Elle permet à l'utilisateur de s'identifier sur le serveur, et ainsi de n'accéder qu'aux exécutions qui lui sont autorisées.

L'image 5.1 illustre la fenêtre d'accueil et la fenêtre de connexion lors du lancement du logiciel.

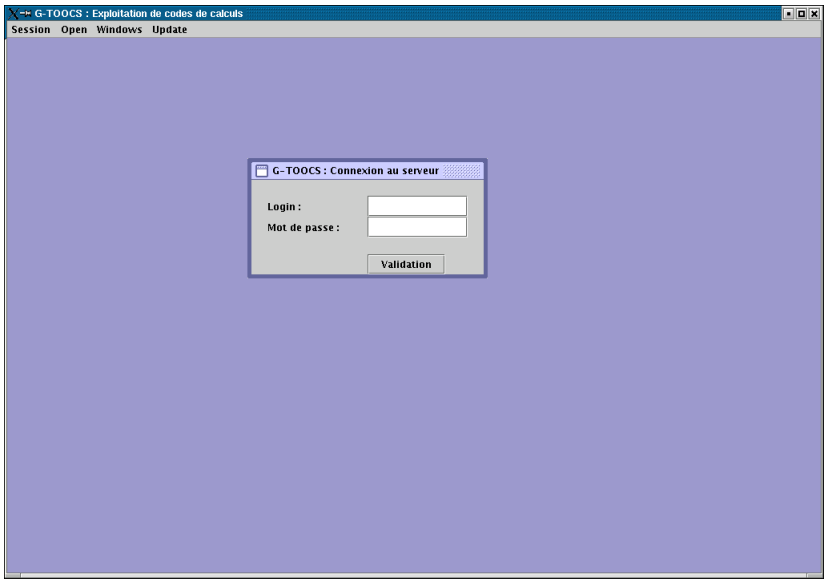

FIG. 5.1 – Fenêtre d'accueil et de connexion de l'interface graphique.

5.3.4 Fenêtre des données utilisateur

Si l'utilisateur est autorisé à se connecter au serveur, deux nouvelles fenêtres d'ouvrent. La première correspond aux données propres à l'utilisateur connecté. Elle lui offre en outre la possibilité de changer son mot de passe sur la base des utilisateurs.

5.3.5 Fenêtre des calculs

La seconde fenêtre qui s'ouvre lorsque la connexion est réalisée est celle qui comporte, dans un tableau, la liste des exécutions recensées sur le serveur.

L'image 5.2 présente la fenêtre utilisateur, ainsi que la fenêtre des calculs.

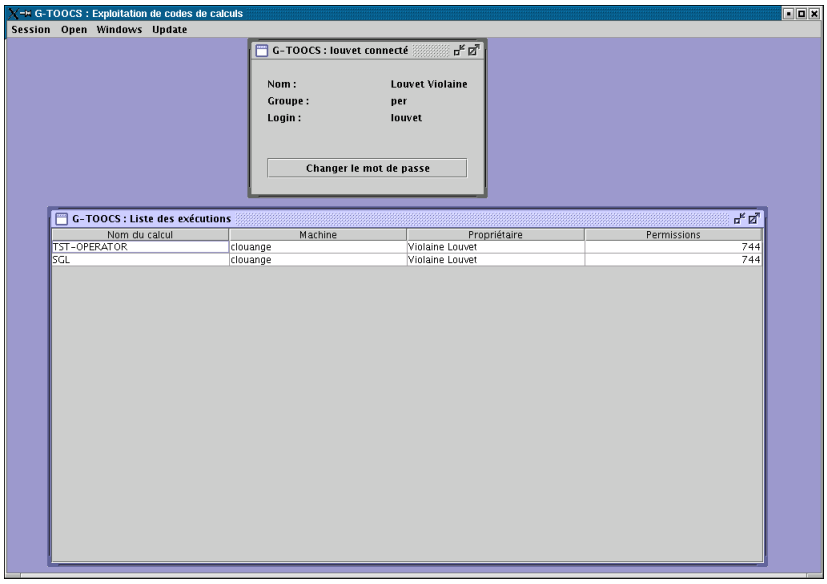
G-TOOCS : Exploitation de codes de calculs
Session Open Windows Update
G-TOOCS : louvet connecté
Nom : Louvet Violaine
Groupe : per
Login : louvet
Changer le mot de passe
G-TOOCS : Liste des exécutions
Nom du calcul Machine Propriétaire Permissions
TST-OPERATOR clouange Violaine Louvet 744
SGL clouange Violaine Louvet 744

FIG. 5.2 – Fenêtre donnant les informations de l'utilisateur connecté et des calculs accessibles par l'interface graphique.

5.3.6 Fenêtre d'une exécution particulière

La sélection d'une ligne du tableau des exécutions engendre l'ouverture de la fenêtre de l'exécution correspondante. Celle-ci est constituée de cinq parties distinctes :

- la partie informations générales qui recense les données temps réel (machine sur laquelle tourne le calcul, temps d'exécution, état du calcul),
- la partie contrôle qui permet de contrôler l'exécution à distance,
- la partie flux de données qui permet d'avoir accès à chacun des flux définis dans le code,
- la partie documentation, qui permet de visualiser la documentation du code, lorsque celle-ci a été structurée comme il est expliqué dans la partie Documentation du rapport,
- la partie suppression du serveur, qui n'est accessible que lorsque l'exécution est terminée. Cette action permet de supprimer tous les flux distribués du serveur lorsque l'exécution a été créée avec l'option *PERSISTENT*.

L'image 5.3 présente un exemple de fenêtre d'exécution.

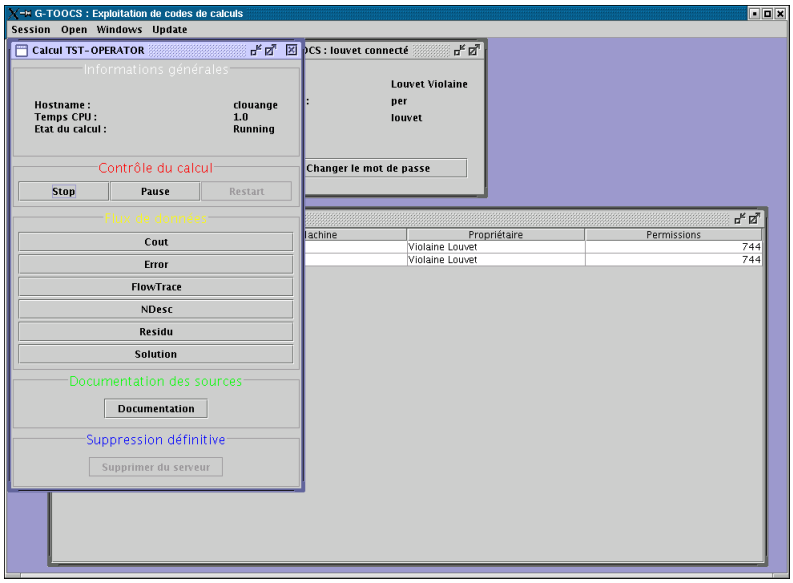

FIG. 5.3 – Fenêtre donnant les informations d'un calcul particulier accessible par l'interface graphique.

5.3.7 Fenêtre d'un flux particulier

L'action sur le bouton d'un flux particulier ouvre la fenêtre correspondante à ce flux. Cette fenêtre comporte un champ rappelant le format du flux, ainsi qu'une barre de menu comprenant les menus suivants :

- menu *Texte* : il comprend les items permettant une visualisation textuelle des données,
- menu *Graphiques* : il comprend les items permettant une visualisation graphique des données,
- menu *Statistiques* : non implémenté.

Selon le format du flux, certains items ne sont pas accessibles.

L'image 5.4 suivante présente plusieurs fenêtres particulières de flux. Une fenêtre unique est ouverte pour chaque flux.

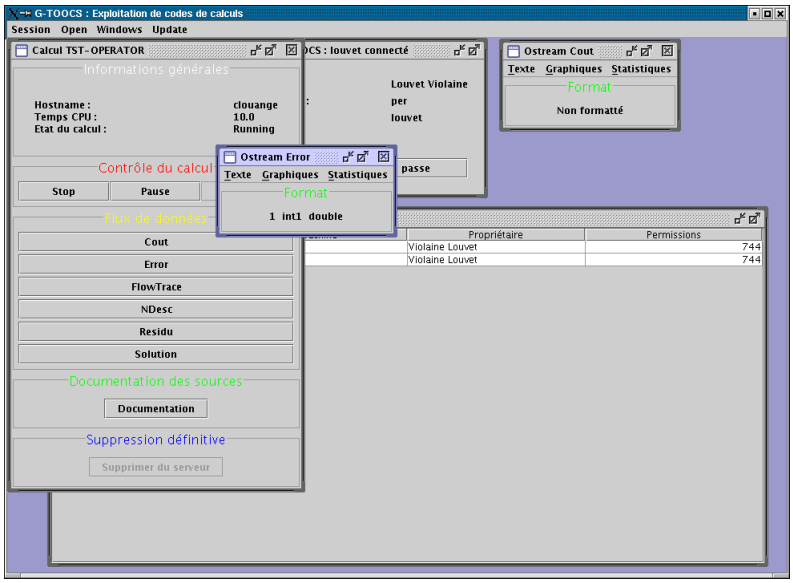

FIG. 5.4 – Fenêtre donnant les informations de flux issus d'un calcul accessible par l'interface graphique.

Visualisation sous format textuel

Tous les flux peuvent être visualisés de façon textuelle. Le menu *Texte* contient deux items :

- *Show All* : permet de visualiser l'ensemble des enregistrements de ce flux,
- *Show Last Records* : permet de visualiser les derniers enregistrements de ce flux. Une fenêtre de dialogue permet d'indiquer le nombre d'enregistrements souhaités.

L'exemple 5.5 illustre la visualisation du flux associé à la variable *Cout*.

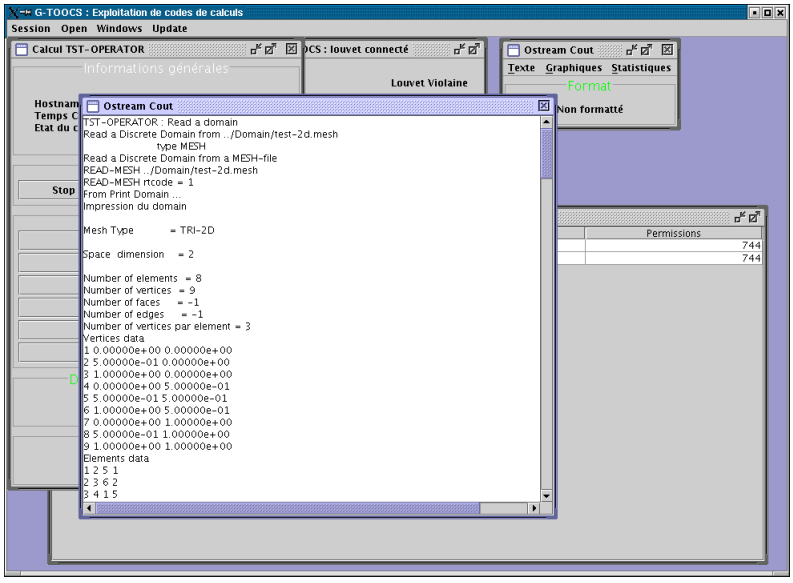

FIG. 5.5 – Fenêtre permettant la visualisation textuelle d'un flux par l'interface graphique.

Visualisation sous format graphique

L'accès aux items du menu *Graphiques* n'est possible que pour les flux formattés. Le menu *Graphiques* contient deux sous-menus :

- *2D* : pour le tracé de courbes en 2 dimensions,
- *3D* : pour le tracé de courbes en 3 dimensions, non encore implémenté.

Chacun de ces sous-menus comporte deux items :

- *Courbe sur tous les enregistrements* : lorsque chaque élément du flux est de taille 1, il n'est pas possible de tracer une courbe avec une seule valeur. La seule possibilité est donc de considérer l'unique valeur de tous les enregistrements. Ce choix est indisponible dans le cas contraire.
- *Courbe sur le dernier enregistrement* : lorsque chaque élément du flux est de taille > 1, en général de grande taille, il n'est possible de tracer une courbe qu'avec un seul enregistrement. Ce choix est indisponible dans le cas contraire.

Le choix d'un de ces items ouvre la fenêtre de choix des données pour le tracé de la courbe. L'image 5.6 en est un exemple.

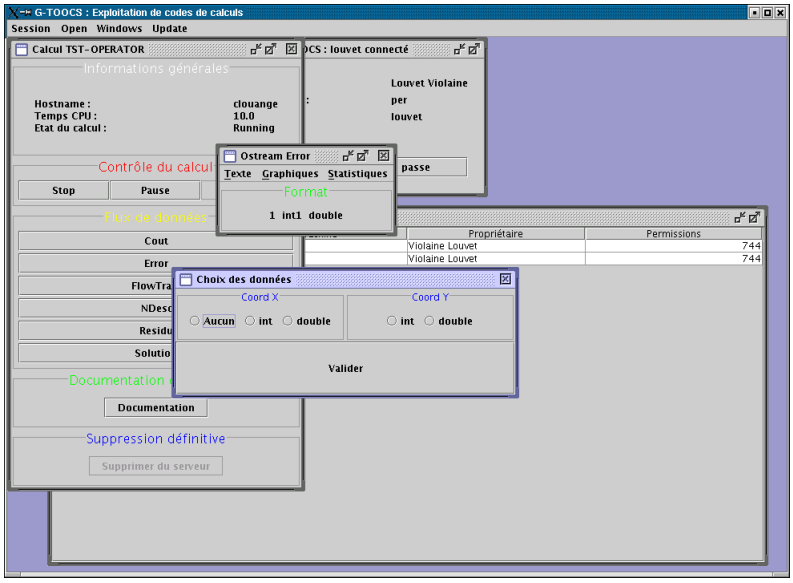

FIG. 5.6 – Fenêtre permettant le choix des données pour la visualisation graphique d'un flux via l'interface graphique.

Pour une courbe 2D, il faut donc indiquer :

- les données en abscisses : elles peuvent être simplement les indices du tableau des données en ordonnées, dans ce cas il faut choisir *"Aucun"*, ou un des éléments du flux,
- les données en ordonnées : il faut choisir un des éléments du flux.

Dans le cadre de notre exemple, l'élément *entier* du flux *Error* correspond à l'itération, et l'élément *double* correspond à l'erreur pour cette itération. Il est donc logique de choisir en abscisse le numéro de l'itération, donc l'élément *entier*, et en ordonnée la valeur de l'erreur, donc l'élément *double*.

Ce choix conduit au tracé de la courbe de l'erreur par rapport aux itérations illustrée par la figure 5.7.

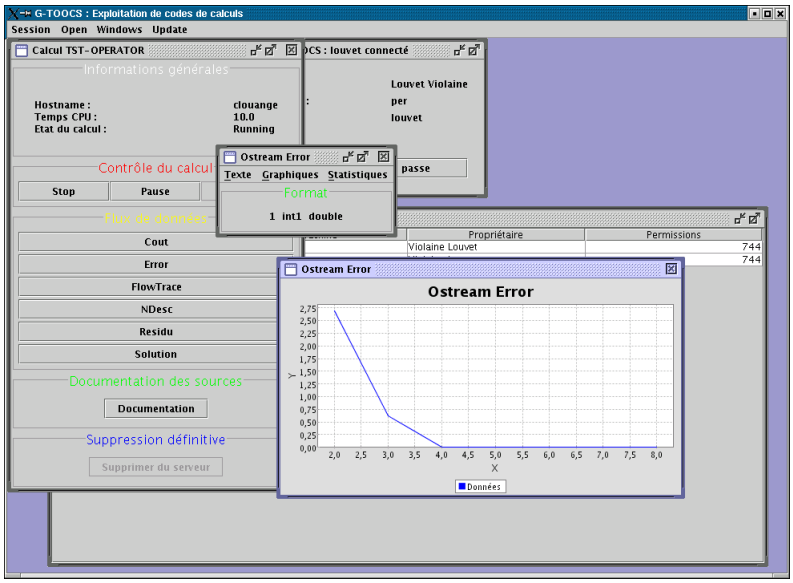

FIG. 5.7 – Fenêtre permettant la visualisation graphique (courbe 2D) d'un flux via l'interface graphique.

5.3.8 Fenêtre de documentation

Il est possible, depuis l'interface graphique d'exploitation, d'accéder à la documentation des sources des codes instrumentés. L'action sur le bouton *Documentation* de la fenêtre d'une exécution ouvre une fenêtre de choix de fichiers, où sont filtrés les fichiers sources (c'est-à-dire d'extension .h, .cpp, .f90, .java, .idl ...).

Si les sources des codes instrumentés ont été correctement structurés au niveau de leur documentation, le choix d'un de ces fichiers permet la génération automatique de cette documentation et sa visualisation dans l'interface graphique.

L'image 5.8 illustre cette possibilité.

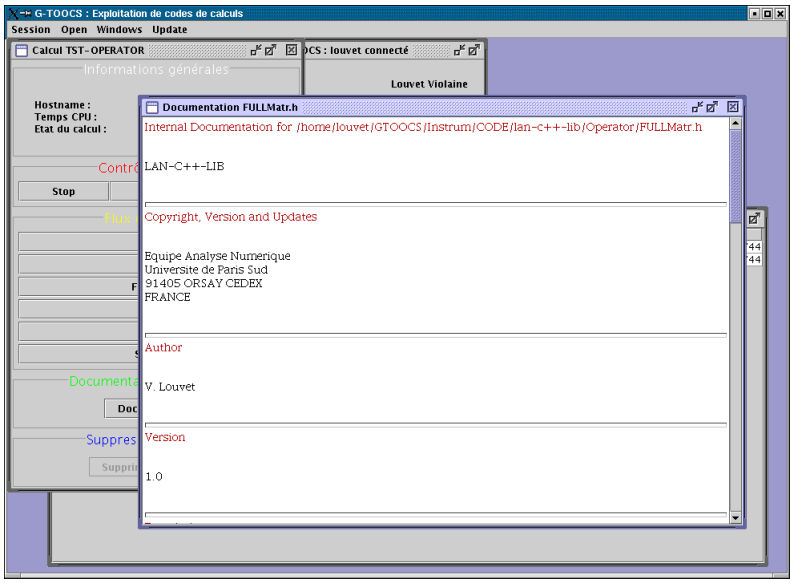

FIG. 5.8 – Fenêtre permettant la visualisation de la documentation d'un code via l'interface graphique.

Il faut noter qu'il existe un outil indépendant de l'interface d'exploitation pour la génération et la visualisation de la documentation des sources d'un code. Cet outil est décrit dans la partie Documentation de ce rapport.

Chapitre 6

Protocole de documentation

Introduction

L'objectif du projet GT-OOCS est de mettre à disposition des développeurs de codes de calcul scientifique un environnement de programmation complet :

- aide à la conception oreintée objet,
- aide au développement,
- possibilité d'exploitation, de contrôle et de couplage du code.

Cet environnement est complété par la mise à disposition d'une structure de documentation pour les sources des programmes informatiques.

Cette structure s'accompagne :

- d'un outil, *makedoc*, permettant la génération de la documentation HTML associée,
- d'une interface graphique facilitant l'utilisation de cet outil.

Makedoc est un outil d'extraction automatique de la documentation relative à un programme informatique. Les langages disponibles dans la version actuelle sont les suivants :

- Fortran 90,
- C++,
- perl,
- idl,
- Java.

Il fonctionne à l'aide de mots-clés, introduit sous forme de commentaires dans les programmes, permettant de décomposer la documentation en différents paragraphes.

La documentation est générée directement à partir des lignes actives du programme et a donc l'intérêt primordial d'être forcément à jour à chaque fois qu'elle est demandée.

Plusieurs niveaux de documentation peuvent être extraits, selon que l'on destine la documentation à un développeur utilisant le programme ou à un programmeur intervenant sur le programme.

Le programme d'extraction est écrit en perl. La documentation est générée en html.

Dans le cas du C++, on extrait la documentation du *.h et du *.cpp associé dans le même fichier *.html.

Cet outil a été initialement développé dans le cadre de la bibliothèque Fortran 90 orientée objet mise en oeuvre au Laboratoire de Mathématique d'Orsay ([**?**]).

6.1 Niveau de documentation

Nous définissons deux niveaux de documentation :

1. Le niveau utilisateur que l'on notera level1,
2. Le niveau programmeur que l'on notera level2.

Chacun des mots-clés permettant l'extraction de la documentation va correspondre à l'un de ces niveaux. Makedoc permet ainsi de générer une documentation au cas par cas. Un programme bien documenté permettra donc de donner les explications nécessaires :

1. à la description du contenu du programme : ce sont les commentaires de niveau 1 (level1) donnant des informations générales (objectifs, fonctionnement, includes nécessaires, sous-programmes appelés ...),
2. à la compréhension des algorithmes implantés : il s'agit alors de comentaires techniques de niveau 2 (level2).

Tous les commentaires se trouvant en niveau 1 de documentation (c'est-à-dire le niveau utilisateur), se trouvent également par défaut en niveau 2 (c'est-à-dire programmeur). A chaque description de mot-clé ci-dessous se trouve associé un niveau d'extration de documentation.

Cette documentation se veut interactive, c'est-à-dire qu'elle est à jour à chaque modification du programme. Elle ne doit pas être un doublon des commentaires propres au programme : elle doit être ces commentaires.

6.2 Structuration de la documentation

On distingue deux parties dans la documentation :
– L'entête,
– Le corps du programme.

Aucun mot-clé n'est obligatoire, exceptés les mots-clés signalant le début et la fin de la documentation :

doc qui signale le début de la documentation : c'est de cette partie que l'information constituant la documentation au format HTML est extraite.

end qui en signale la fin.

Ces deux mots-clés se trouvent dans les niveaux 1 et 2 de la documentation (level1 et level2).

6.2.1 Structure de l'entête

L'entête va servir à commenter les informations générales du programme. Pour cela, on définit un certain nombre de mots-clés qui permettent de structurer ces informations :

doc (level1) comme on l'a déjà vu, indique le début de la documentation. Il peut être suivi du nom de la bibliothèque à laquelle est rattaché le programme.

copyright (level1) indique le paragraphe du copyright.

author (level1) indique l'auteur du programme.

version (level1) donne le numéro de version en cours à mettre à jour à chaque changement.

description (level1) décrit le rôle du fichier.

modifications (level2) indiquent les modifications qui ont été effectuées sur le fichier, avec la date, la version correspondante, l'auteur et la nature des modifications.

todo (level2) donne les futurs développements à effectuer.

6.2.2 Structure du corps du programme

Le programme est commenté par les mots-clés suivants, qui permettent de lire directement les informations dans le code :

include (level1) indique les noms des fichiers utilisés par le fichier courant.

name (level1) donne le nom de la routine/classe/module.

parameters (level1) décrit les déclarations et définitions des paramètres passés en argument d'une routine.

variables (level1) donne les déclarations et définitions des variables locales.

purpose (level1) permet la description de l'entité associée à \name.

calling_sequence (level1) décrit la séquence d'appel de la routine en relation avec \parameters.

called (level1) indique les fonctions appelées par la routine (langage F77 et C où il n'y a pas d'include)

level1 (level1) correspond à la documentation de niveau 1 (par défaut).

level2 (level2) correspond à la documentation de niveau 2.

skip (level1) indique que l'on passe cette partie du programme. Pour reprendre la génération de documentation après un skip, on utilise les mots-clés définis ci-dessus et on se trouve en level 1 par défaut ou on utilise d'abord \level2 pour indiquer une documentation de niveau 2.

6.3 Utilisation du programme

6.3.1 Mode en ligne

Le programme fonctionne selon deux types d'arguments passés sur la ligne de commande :
- On passe en arguments le nom court du fichier (sans extension), le langage et le niveau (optionnel) : *makedoc fichier f90 1*
- On passe en arguments le nom complet du fichier : *makedoc fichier.f90*

Les mots clés désignant les langages sont les suivants :
- f90 pour le fortran 90,
- perl ou pl pour le perl,
- cpp ou cc pour le C++,
- java pour le Java,
- idl pour l'IDL

Les extensions des fichiers doivent être standards à savoir :
- .f90,
- .pl,
- .cpp,
- .java,
- .idl.

6.3.2 Mode graphique

Une interface graphique est disponible pour faciliter l'utilisation de l'outil *makedoc*.

Elle est constituée d'une fenêtre proposant l'ouverture d'un fichier à partir des arbres des répertoires du système et selon un filtre portant sur les extensions des fichiers suivant les langages traités par *makedoc* (.pl, .cpp, .java, .idl, .f90).

La fenêtre permettant le choix du fichier est illustrée par la figure 6.1.

La sélection d'un fichier entraîne la création du fichier HTML associé et l'ouverture d'un browser HTML sur le fichier généré.

La documentation issue d'un fichier source est donc directement disponible à l'écran.

La fenêtre permettant la visualisation du code HTML est présentée dans les exemples ci-dessous.

6.4 Exemples

Deux exemples sont présentés dans cette section. L'un est issu d'un code source en fortran 90, et l'autre d'un code source en C++.

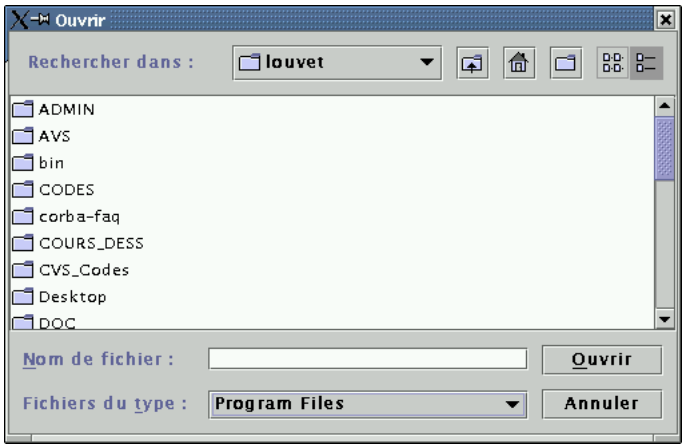

FIG. 6.1 – Fenêtre permettant le choix du fichier pour la génération de la documentation

6.4.1 Exemple en fortran 90

Programme documenté

Le programme fortran 90 documenté selon la structure supportée par *makedoc* est donné par le source suivant.

```
!\doc

! B-EF-F-OO : Bibliotheque d'Elements Finis Fortran-90 Orientee Objet

!\copyright

! Equipe Analyse Numerique
! Universite de Paris Sud
! 91405 ORSAY CEDEX
! FRANCE

!\author

! V.LOUVET

!\version

! 0.1

!\description

! GEOM_LIBR : Module of the interface of geom

!\modifications
!  02/2000   0.1   V. Louvet   Création

!\name

module geom_libr

!\include
```

```
  use contxt_desc  ! General purpose Module
  use contxt_libr  ! Input/ouput toolkit Module
  use ftrace_libr  ! Flow-Trace Module

  use geom_desc    ! Geom Module

!\skip

  interface

!\calling_sequence

     subroutine gm_rect_2d(geom, &
          x1, x2, y1, y2, &
          nbx, nby)

!\skip

       use contxt_desc  ! General purpose Module
       use contxt_libr  ! Input/ouput toolkit Module
       use storag_desc  ! Storage toolkit     Module
       use ftrace_libr  ! Flow Trace         Module
       use geom_desc    ! Mesh Descriptor     Module
       implicit none
       type(geom_struc), intent(inout) :: geom   ! The mesh
       integer         , intent(in   ) :: nbx ! Number of segements beetwing S1 S2
       integer         , intent(in   ) :: nby ! Number of segements beetwing S1 S3
       real(rp)        , intent(in   ) :: x1, y1 ! Coordonnates of S1
       real(rp)        , intent(in   ) :: x2, y2 ! Coordonnates of S2
     end subroutine gm_rect_2d

  end interface

end module geom_libr

!\end
```

Documentation générée

Les figures 6.2 et 6.3 présentent la visualisation du code HTML généré par *makedoc* à partir du code source du programme fortran 90 ci-dessus.

6.4.2 Exemple en C++

Programme documenté

Le code C++ considéré est constitué d'un source d'entête *.h et de l'implémentation correspondante *.cpp donnés ci-dessous.

Fichier PingServer_impl.h :

```
//\doc
```

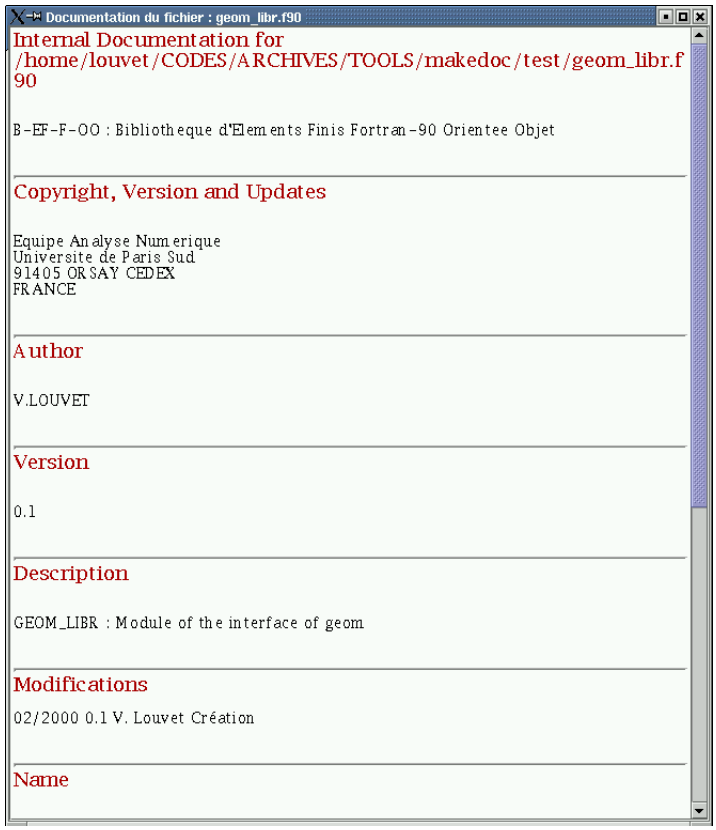

FIG. 6.2 – Visualisation de la documentation générée pour le code source en fortran 90 : 1ère partie

```
// G-TOOCS

//\copyright
//     Equipe Analyse Numerique
//     Universite de Paris Sud
//     91405 ORSAY CEDEX
//     FRANCE

//\author
// V. Louvet

//\version
// 1.0

//\description
//     Classe servant implementant l'interface CORBA Ping
//     pour le test de la connexion au serveur
//     Classe Singleton.

//\modifications
//      31/12/02     1.0      V. Louvet     Creation

//\skip
```

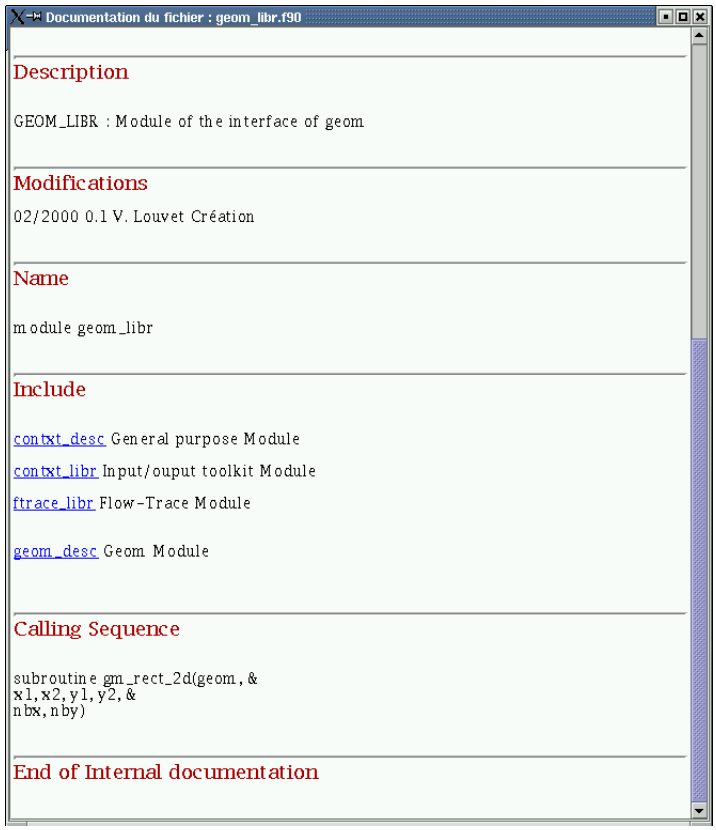

FIG. 6.3 – Visualisation de la documentation générée pour le code source en fortran 90 : 2ème partie

```
#ifndef PINGSERVER_IMPL_H
#define PINGSERVER_IMPL_H

using namespace std;

//\include
#include <OB/CORBA.h>
#include "PingServer_skel.h"

//\name
class PingServer_impl : virtual public POA_PingServer
//\skip
{

 private:

...

 protected:

  // Constructor
  PingServer_impl();
```

```
 public:

  //\calling_sequence
  // Unique instance of the class
  static void instance();

  //\calling_sequence
  // Destructor
  ~PingServer_impl();

  //\calling_sequence
  // Pinging the server
  virtual void ping()
    throw(CORBA::SystemException);

  //\calling_sequence
  // End of the server
  static void end();

  //\skip
};
#endif //PINGSERVER_IMPL_H
```

Fichier PingServer_impl.cpp :

```
//\doc
//\skip

#include "PingServer_impl.h"
#include "CorbaServer.h"

// ----------------------------------------------------------------------
// Null initialization of static members
PingServer_var PingServer_impl::_instance = 0;
PingServer_impl* PingServer_impl::_instanceImpl = 0;

// ----------------------------------------------------------------------
// Static method to declared the unique instance of the class
void PingServer_impl::instance() {
...
}

// ----------------------------------------------------------------------
// Constructor
PingServer_impl::PingServer_impl()
{}

// ----------------------------------------------------------------------
// Destructor
PingServer_impl::~PingServer_impl() {
...
}
```

```
// --------------------------------------------------------------------
// Pinging the server
void PingServer_impl::ping()
  throw(CORBA::SystemException)
{}

// --------------------------------------------------------------------
// End of the server
void PingServer_impl::end() {
...
}

//\end
```

Documentation générée

Dans le cas du C++, *makedoc* génère une documentation globale pour le fichier entête et le fichier implémentation correspondant. Les figures 6.4 et 6.5 illustrent la visualisation HTML de cette documentation.

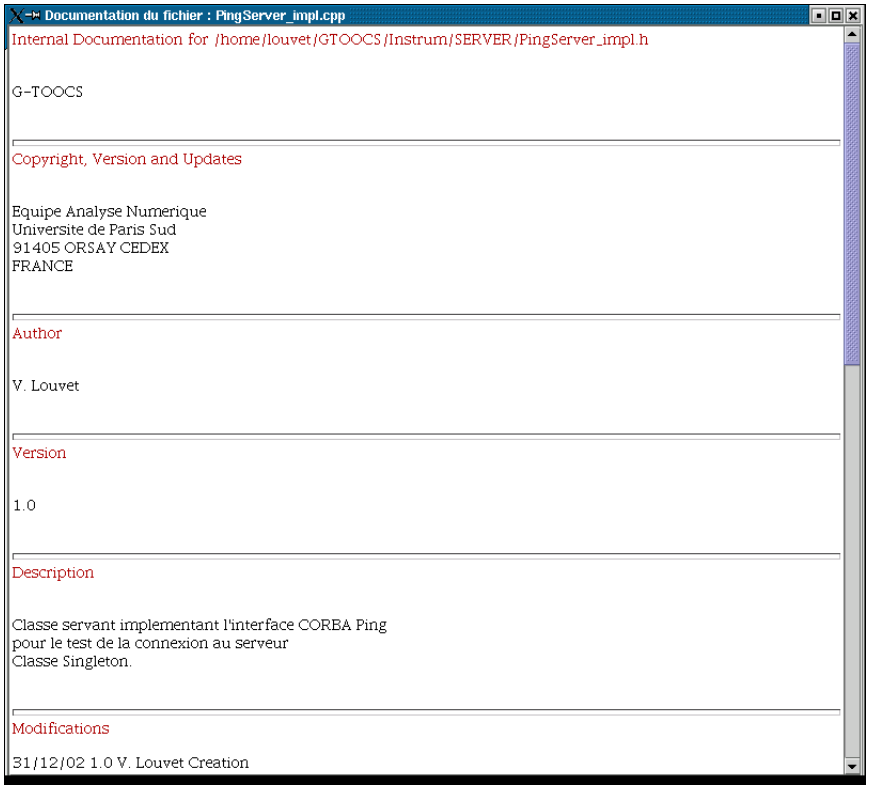

FIG. 6.4 – Visualisation de la documentation générée pour le code source en C++ : 1ère partie

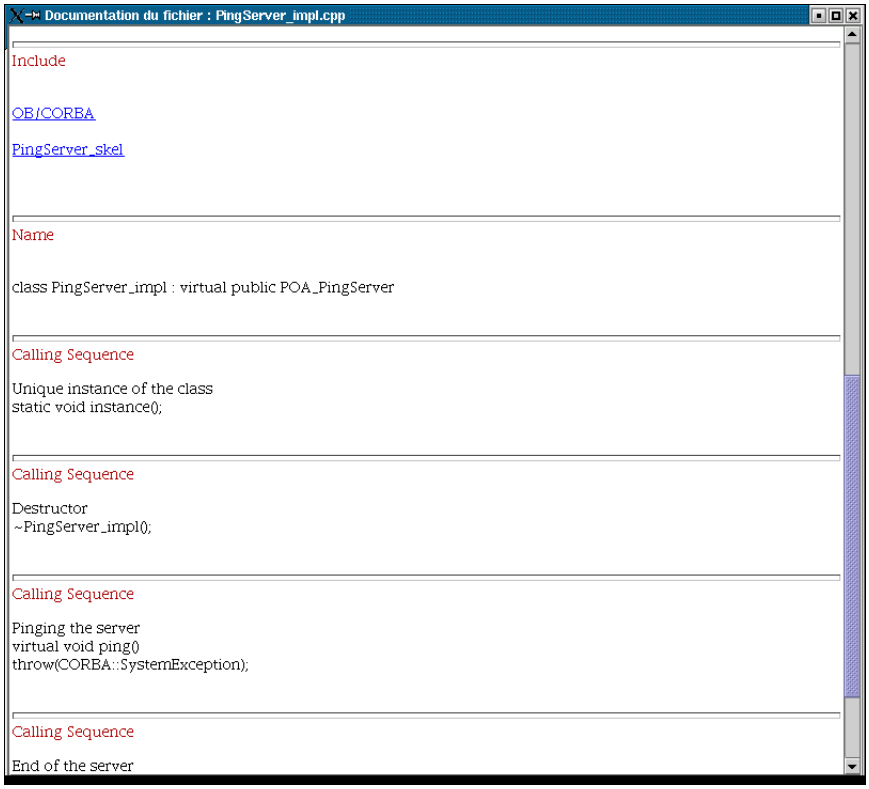

FIG. 6.5 – Visualisation de la documentation générée pour le code source en C++ : 2ème partie

Annexe A

Etat de l'art en bibliothèques et outils de calcul scientifique

Deux aspects sont développés ici : les projets analogues au notre et les bibliothèques de calcul scientifiques existantes. La liste des travaux reliés à MOOCS est longue. Le but de ce chapitre n'est pas d'en faire la liste complète, mais de sélectionner certains projets ou réalisations en fonctions de certains critères que nous vous proposons de définir (paragraphe A.1) puis les paragraphes suivants proposeront une brève description de ces travaux.

A.1 Les critères

Le premier critère est celui de la disponibilité des bibliothèques ou logiciels. Ce projet de par sa définition n'a de sens que s'il reste dans le domaine public et donc l'ensemble des éléments qui nous sert de base ou de composants doivent être également disponible pour la recherche. Donc ne seront examinés que les outils du domaine publique ou ceux pouvant faire l'objet d'un contrat de partenariat au titre de la recherche, ceci sans mettre en doute l'exellence de très nombreux travaux du domaine commercial. Une exception sera faite pour les utilitaires graphiques. **AVS/EXPRESS** est un logiciel ouvert que notre équipe possède depuis lontemps et pour lequel l'écriture d'interfaces sera envisagée. Par ailleurs parmi les travaux du domaine publique nous ne retiendrons que ceux dont le code source est disponible (ce qui est en général le cas), ceci nous assurant une plus grande portabilité (re-compilation, adaptations mineures, ...)

Les sous-paragraphes suivants étudient au cas par cas certaines catégories de logiciels et bibliothèques. On distinguera ceux qui ont une structuration objet et les autres. Il est clair que tout ne peu être passé en revu. D'autre part nous sommes plus concernés par la structure des logiciels et des bibliothèques que par leurs contenus.

De nombreux logiciels et bibliothèques proviennent de la "NETLIB" (`http ://www.netlib.org`) et de "GNU"(`http ://www.gnu.org`).

Ce chapitre n'est pas un répertoire de TOUT ce qui existe. Ce que nous cherchons dans les bibliothèques et les logiciels existants, c'est essentiellement le savoir-faire, les principes de développement qui ont été retenus et surtout le pourquoi.

A.2 Logiciels et Bibliothèques Objets

Nous avons vu que l'approche objet est un critère essentiel de nos développements. Ce que nous recherchons dans les codes existants, c'est justement cette analyse objet. Comprendre comment on est amené à choisir la définition d'une classe est très important. Si la méthodologie procédurale conduisait finalement à une structure de code à peu prés identique, il n'en est rien des bibliothèques objets.

On peut distinguer deux types d'objets, les objets définissant la structure globale de code et les objets "utilitaires". Les premiers permettent d'exprimer le problème à résoudre et les algorithmes de résolutions. Il dépend fortement de la philosophie objet utilisée. Il n'est donc pas évident que deux philosophies différentes aboutissent sur des objets compatibles. Par contre les objets "utilitaires" comme la gestion des tableaux, des matrices, le traitement de erreurs, etc, interviennent finalement dans des couches plus profondes de l'écriture et peuvent aisement être compatibles entre des philosophies différentes.

A.2.1 elsA

http ://elsA.onera.fr

Le projet elSa (Ensemble Logiciel pour la Simulation en Aérodynamique) est développé à l'ONERA. C'est un logiciel effectivement conçu de façon objet, très proche du notre dans sa philosophie.

A.2.2 FreeFEM++

http ://www.freefem.org

FreeFEM++, comme son nom l'indique est un logiciel de résolution de problèmes aux dérivées partielles discrétisées par éléments finis. Il a été conçu par F. Hecht, O. Pironneau (Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, Paris) et K. Ohtsuka (Hiroshima Kokusai Gakuin University, Japan) et est écrit en C++. Il constitue en réalité un méta-langage.

Il comprend un générateur de maillage avec des possibilités d'adaptation. La version 1.29 permet de travailler avec les éléments P1, P1 non conforme, P2 et les éléments de Raviart-Thomas.

A.2.3 Mélina

http ://perso.univ-rennes1.fr/daniel.martin/melina/www/homepage.html

"Mélina est une bibliothèque de procédures pour la résolution de problèmes aux limites gouvernés par des équations aux dérivées partielles par la méthode des éléments finis en dimension 2 ou 3. Il a été développé par O. Debayer (de 1989 à 1996) et par D. Martin (depuis 1989). Ce code est essentiellement un code de recherche et fournit un ensemble d'outils aisément manipulables pour écrire rapidement des applications pour le traitement numérique de problèmes nouveaux" (extrait du guide de l'utilisateur).

Mélina est écrit en Fortran 77. Il est conçu autour d'utilitaires de haut-niveau (macros-opérations).

A.2.4 Overture

http ://www.llnl.gov/CASC/Overture

Développé au Lawrence Livermore National Laboratory, Overture est un ensemble de classes (écrites en C++) pour la représentation de grilles et propose un ensemble de méthodes sur les grilles. Overture utilise intensivement les classes de tableaux (séquentiels et parallèles) A++/P++ et propose des discrétisations par différences finies et volumes finis.

A.3 Logiciels et Bibliothèques NON Objets

De nombreuses bibliothèques du domaine publique proposent des utilitaires d'algèbre linéaire pour la résolution de systèmes linéaires, le calcul de valeurs propres ou singulières, la factorisation de matrices ... Nous ferons une distinction entre les méthodes de résolutions directes et itératives.

La grande majorité des méthodes itératives récentes sont des méthodes de descente pour lesquels la matrice n'intervient que par l'intermédiaire du produit MATRICE-VECTEUR. Par ailleurs ces méthodes sont simples de mise en œuvre ; ce qui n'est pas le cas pour les méthodes directes. On choisit donc de ne s'intéresser qu'aux bibliothèques d'algèbre linéaire dont les algorithmes reposent sur des méthodes directes.

A.3.1 Les bibliothéques d'algèbre linéaire

L'ensemble LINPACK, EISPACK

LINPACK est une collection de sous-programmes FORTRAN-77 pour la résolution de systèmes linéaires et de problèmes linéaires de moindres carrés [**?**]

EISPACK est une collection de sous-programmes FORTRAN-77 qui calculent les valeurs propres et les vecteurs propres pour neuf classes de matrices [**?**].

Ces deux bibliothèques sont basées sur des fonctions d'algèbre linéaire élémentaires de niveau 1 : BLAS-1 [**?**], c'est-à-dire d'opérations sur des tableaux monodimensionnels. Elles sont particulièrement bien adaptées aux calculateurs vectoriels pour lesquels le BLAS-1 pouvait être particulièrement bien optimisé.

L'ensemble BLAS, LAPACK, SCALAPACK

Cet ensemble représente une refonte complète des bibliothèques LINPACK et EISPACK. Par l'utilisation de BLAS de niveau 2 et 3, LAPACK est bien adaptée aux machines à hiérarchie de mémoires (caches). La version d'origine est écrite en FORTRAN-77 mais elle possède des versions en C (CLAPACK), en C++ (LAPACK++) en FORTRAN-9X (LAPACK90) et également une version parallèle (SCALAPACK). De ce fait, nous ferons une étude complète de cet ensemble qui est un **standard de fait** dans le monde de l'algèbre linéaire. De plus, les versions C++ et FORTRAN-90 interviennent dans le cadre d'une réflexion sur l'encapsulation de sous-programmes procéduraux dans des langages objets ou orientés objets.

BLAS est un ensemble de sous-programmes d'algèbre linéaire élémentaires écrits en FORTRAN-77. On distingue le BLAS-1 qui travaille sur des vecteurs (voir LINPACK), le BLAS-2 qui travaille sur des matrices et le BLAS-3 qui est une version par blocs des BLAS de niveaux inférieurs. Les BLAS forment les sous-programmes de base de LAPACK. Des versions "constructeurs" existent, optimisés sur chaque plateforme.

LAPACK est un ensemble de sous-programmes FORTRAN-77 pour la résolution des problèmes d'algèbre linéaire les plus communs : systèmes linéaires, problèmes de moindres carrés linéaires, problèmes de valeurs propres et de décompositions en valeurs singulières [**?**].

CLAPACK est la version en langage C de LAPACK. Elle a été obtenue à partir de la version FORTRAN grâce à `f2c`, un traducteur de FORTRAN vers C (voir GNU). Son intérêt est moindre.

LAPACK90/95 est une encapsulation de la version FORTRAN-77. Deux types d'encapsulations sont proposées.

La première, très proche de LAPACK pour ce qui est des fonctionalités permet de s'affranchir du typage des sous-programmes en fonction de celui des matrices et vecteurs. La notion d'**interface générique** est fortement utilisée.

La seconde version est "plus FORTRAN-90". Elle exploite également la notion d'interface générique, mais on utilise ici d'autres moyens de distinction des sous-programmes comme celui du nombre d'arguments, ...

LAPACK++ est devenu un projet plus ambitieux qu'une simple encapsulation des programmes FORTRAN [**?**]. La problématique des classes de tableaux est posée. Ce point particulier fait d'objet d'un paragraphe spécifique (voir A.4). TNT est l'ensemble des classes "tableaux" utilisées par LAPACK++.

SCALAPACK est une version parallèle de LAPACK adaptée aux machines à mémoire distribuée. Elle utilise la bibliothèques de passage de messages MPI (voir ci-après). SCALAPACK est présentée en plusieurs sous-ensembles dont BLACS la bibliothèque des fonctions élémentaires de passages de messages.

La bibliothèque SCILIB

SCILIB est une bibliothèque FORTRAN-77 portable émulant la bibliothèque CRAY SCILIB (développé par Cray Resarch, Inc.). Elle est dû à M.J. McBride et S.H. Lamson.

A.3.2 Les autres bibliothèques

Outre la résolution de systèmes linéaires, de nombreux autres domaines sont traités dans les bibliothèques de calcul scientifique : fft, fonctions spéciales, fonctions aléatoires,...

A.4 Les classes objets pour la gestion des tableaux

Idéalement on voudrait des fonctionnalités identiques que celles proposées par les langages Fortran-90/95 qui sont des langages de programmation très orientés calcul scientifique, c'est-à-dire :

- des tableaux à plusieurs indices pour les types par défaut (entier, flottant, logique et chaine de caratères) mais aussi pour les types utilisateurs,
- des opérations sur les tableaux (somme, produit, produit scalaire, extraction de sous-tableaux, fusion, ...).

Pour rendre MOOCS indépendant du choix de la notion de tableau, nous devons définir un ensemble de fonctionnalités minimal, et les interfacer par l'intermédiaire des outils disponibles dans ce domaine et décrits ci-dessous.

A.4.1 Le standard de C++

On trouve en C++ le type standard des tableaux sous la forme de pointeurs. Cette solution est particulièrement peu souple, peu modulaire et donc complètement inadaptée à nos besoins.

L'intérêt de la norme du C++ en matière de tableaux résident dans les conteneurs de la bibliothèque standart (STL). Par définition, un conteneur est un objet qui en contient d'autres. Cette généricité est basée sur le paramétrage des types. On peut citer en exemple les listes, les vecteurs, les tableaux associatifs ... qui peuvent contenir des types fondamentaux (entier, double ...) ou des types utilisateur.

Chacun des conteneurs de la STL a été conçu de façon optimale pour l'approche qu'il modélise ; ainsi, un `vector` est optimisé pour les accès aléatoires, tandis qu'une `list` le sera pour les opérations sur les éléments de début et de fin.

La bibliothèque standard contient aussi une structure de données génériques, que l'on peut qualifier de "presque conteneur" ([**?**]) car elle n'interface pas complétement tous les aspects des conteneurs. Il s'agit du `valarray`, qui est un vecteur destiné aux traitements numériques optimisés. C'est cette structure qu'il faut retenir pour la manipulation de tableaux standards en C++. Par contre, il faut noter que le multi-indiçage n'est pas immédiat et nécessite l'interfaçage de la structure.

A.4.2 Le standard de JAVA

Java gère les tableaux de type intégré et de référence, avec des lancements d'exceptions lors des accès hors-borne. Il est possible ainsi de définir des tableaux multidimensionnels. Par contre, la taille de ces tableaux est fixe.Un grand nombre d'API Java permettent également de manipuler des collections d'objets (`Array`, `Vector` ...). Il est alors nécessaire de bien cibler le choix de la collection pour assurer des performances optimales.

A.4.3 TNT

C'est l'ensemble des classes de tableaux utilisées par LAPACK++ (voir A.3). Il propose, sous forme de template, des tableaux jusqu'à trois indices et fait la différences entre les tableaux à la mode C et ceux à la mode Fortan. Il offre aussi des possibilités d'opérations sur les tableaux.

A.4.4 A++/P++

Cet ensemble est plus ancien que TNT. Il est conçu sans la notion de template mais possède une version séquentielle A++ et une version parallèle P++. Il a été écrit à Livermore et est fortement utilisé dans Overture. La notion de tableau qui est proposée est très proche de celle de Fortran-90/95. On y retrouve des sections de tableau, des tableaux jusqu'à l'ordre 4 et des opérations sur ceux-ci.

A.5 Les mailleurs

On distingue les grilles propres aux discrétisations par différences finies, les maillages structurés et non structurés pour les éléments finis et les maillages duaux des volumes finis. Il est relativement facile de générer des grilles et des maillages réguliers. Pour les grilles, il faut quand même signaler le problème des domaines de calcul obtenus par une déformation admissible aux sens des différences finies (maillages en C autour d'un profil, maillages en O et leurs combinaisons par exemple). Les maillages réguliers sont intégrés au projet MOOCS. On ne s'intéresse donc qu'aux maillages non structurés. Les maillages duaux sont généralement obtenus à partir de maillages directs. On distinguera le cas des mailleurs 2D (par exemple les mailleurs de MODULEF, ou le projet GAMMA développés par l'INRIA) et celui des mailleurs 3D (par exemple gmsh, http ://www.geuz.org/gmsh).

Les liens des mailleurs (extérieurs au code) avec le code de calcul sont "faibles" dans le sens qu'ils ne sont pas directement intégrés au code, mais les maillages sont transmis soit par l'intermédiaire de fichiers soit par la procédure de couplage de code (voir le paragraphe 3).

On s'intéresse aussi aux outils d'adaptations de maillages et de décompositions, ceux-ci pouvant être directement appelés par le programme de calcul et donc plus ou moins intégrés au code.

A.6 Les bibliothèques d'outils parallèles

On distingue deux types de bibliothèques parallèles : les bibliothèques de passage de messages, et les bibliothèques de synchronisation.

A.6.1 Bibliothèques de synchronisation

Ces bibliothèques sont apparues avec la gamme des machines vectorielles CRAY multiprocesseurs et ne sont généralisées que sur ce type de matériel, c'est-à-dire sur les machines multiprocesseurs à mémoires partagées. Elles proposent des outils de multi-tâches à gros grains pour une programmation SPMD (typiquement un sous-programme) rassemblés en trois groupes :

Gestions des tâches : Il s'agit de permettre à une tâche d'en "lancer" une autre (`call tsk_start(...)`) et d'attendre la fin de son exécution (`call tsk_wait(...)`). Ceci permet une programmation du type "fork and join". Les arguments de ces sous-programmes servent à définir la nouvelle tâche à exécuter et à identifier les couples `tsk_start-tsk_wait`.

Gestions des sections critiques : Une section critique est liée à la notion de variables globales (connues de toutes les tâches) et de variables locales. La section critique définit une portion de code où un accès concurrentiel est fait à une variable globale. La gestion des sections critiques permet de donner une cohérence déterministe au code.

Ces sections sont protégées par des verrous (lock). On peut positionner un verrou (`call lock_on(...)`). Dans ce cas, la première tâche atteignant ce verrou passe, les autres attendent que cette première tâche soit sortie de la section critique (`call lock_off(...)`). On a ainsi une exécution séquentielle des sections critiques.

Gestions d'évènements : Un évènement permet d'envoyer un signal à un ensemble de tâches. On "poste" un évènement (`call even_post(...)`) qui est attendu (`call even_wait(...)`) par d'autres tâches.

D'autres fonctions telles que des barrières existent dans ce type de bibliothèques de synchronisation. Des systèmes de gestions de tâches par liste d'attente ont été programmés de cette façon (voir [?], [?] et [?]).

A.6.2 Les bibliothèques de passages de messages

Les bibliothèques de passages de messages sont adaptées à la programmation de code parallèle pour des machines à mémoires distribuées, bien que des implémentations aient été faites pour des machines à mémoire partagée (voir mixte). Les codes de calcul peuvent être du type MIMD, mais la plupart du temps ils sont SPMD. Le principe est d'envoyer des messages et d'en attendre. Les synchronisations et les tranferts d'informations sont réalisés par ce biais. Il est nécessaire de citer les deux principales bibliothèques que sont PVM (Parallel Virtual Machine) et MPI (Message Passing Interface). Il en existe d'autre, notament P4 sur laquelle est construite la version publique de MPI. Notons également que de nombreux constructeurs proposent des versions optimisées des bibliothèques PVM et MPI bien que disposant de leurs propres bibliothèques.

PVM et MPI ont un mode de fonctionnement différent. PVM possède des fonctions de gestion de type console et a deux modes de lancement des processus, soit par la console soit classiquement. MPI ne possède pas de fonction de gestion et les processus ne peuvent être lancés que par l'intermédiaire d'un programme spécial (`mpirun`).

PVM comme MPI est écrit en C, des interfaces ont été développées pour Fortran-77 (PVM et MPI), Fortran-9x et C++ (MPI) ; pour PVM nous avons développé notre propre interface.

PVM-3

Chronologiquement PVM fut la première bibliothèque de passages de messages. Dans sa première version elle ne proposait que des sous-programmes de gestions de "buffer", d'envoi et de réception. Puis les versions suivantes ont offert la notion de groupes et des fonctions plus proches des besoins des utilisateurs. PVM est, depuis la première version, multi-plateformes et intègre une gestion dynamique des processus.

MPI

Cette bibliothèque est de conception très proche des besoins des utilisateurs. L'ensemble des fonctionalités qu'elle propose est directement utilisable et très riche. MPI devient le standard dans sa catégorie au détriment de PVM.

Annexe B

Notations UML

Introduction

La notation UML (Unified Modeling Language) a été développée en réponse à l'appel à propositions lancé par l'OMG (Object Management Group) dans le but de définir la notation standard pour la modélisation des applications construites à l'aide d'objets.

La notation UML représente l'état de l'art des langages de modélisation objet. Elle se place comme le successeur naturel des notations des méthodes de Booch, OMT (Object Modeling Technique) et OOSE (Object Oriented Software Engineering) et, de ce fait, UML s'est très rapidement imposée, à la fois auprès des utilisateurs et sur le terrain de la normalisation.

Les créateurs d'UML insistent tout particulièrement sur le fait que la notation UML est un langage de modélisation objet et non pas une méthode objet.

Un modèle est une description abstraite d'un système ou d'un processus, une représentation simplifiée qui permet de comprendre et de simuler. Le terme modélisation est souvent employé comme synonyme d'analyse, c'est-à-dire de décomposition en éléments simples, plus faciles à comprendre. En informatique, la modélisation consiste tout d'abord à décrire un problème, puis à décrire la solution de ce problème ; ces activités s'appellent respectivement l'analyse et la conception.

Ainsi, un modèle est une abstraction de la réalité. L'abstraction est l'un des piliers de l'approche objet. Il s'agit d'un processus qui consiste à identifier les caractéristiques intéressantes d'une entité en vue d'une utilisation précise. L'abstraction désigne aussi le résultat de ce processus, c'est-à-dire l'ensemble des caractéristiques essentielles d'une entité, retenues par un observateur.

Un modèle est une vue subjective, mais pertinente, de la réalité. Un modèle définit une frontière entre la réalité et la perspective de l'observateur. Ce n'est pas la "réalité", mais une vue très subjective de la réalité. Bien qu'un modèle ne représente pas une réalité absolue, un modèle reflète des aspects importants de la réalité, il en donne donc une vue juste et pertinente.

Le caractère abstrait d'un modèle doit notamment permettre de faciliter la compréhension du système étudié. Il réduit la complexité du système étudié, permet de simuler le système, le représente et reproduit ses comportements. Concrètement, un modèle réduit (décompose) la réalité, dans le but de disposer d'éléments de travail exploitables par des moyens mathématiques ou informatiques.

UML permet donc de modéliser une application selon une vision objet.

UML définit plusieurs modèles pour la représentation des systèmes :
- le modèle des classes qui capture la structure statique,
- le modèle des états qui exprime le comportement dynamique des objets,
- le modèle des cas d'utilisation qui décrit les besoins de l'utilisateur,
- le modèle d'interaction qui représente les scénarios et les flots de messages,
- le modèle de réalisation qui montre les unités de travail,
- le modèle de déploiement qui précise la répartition des processus.

Les modèles sont regardés et manipulés par les utilisateurs au moyen de vues graphiques, véritables projections au travers des éléments de modélisation contenus par un ou plusieurs modèles. De nombreuses vues peuvent être construites à partir des modèles de base ; elles peuvent montrer tout ou partie des modèles. A chaque vue correspondent un ou plusieurs diagrammes. UML définit neuf types de diagrammes différents :
- les diagrammes de classes,
- les diagrammes de séquence,
- les diagrammes de collaboration,
- les diagramme d'objets,
- les diagrammes d'états-transitions,
- les diagrammes d'activités,
- les diagrammes de cas d'utilisation,
- les diagrammes de composants,
- les diagrammes de déploiement.

Nous allons nous limiter ici à décrire les diagrammes de classes qui ont été utilisés pour la modélisation des développements du projet et la rédaction de ce document.

B.1 Paquetages

Les paquetages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de modélisation. Chaque paquetage est représenté graphiquement par un dossier comme illustré par la figure B.1.

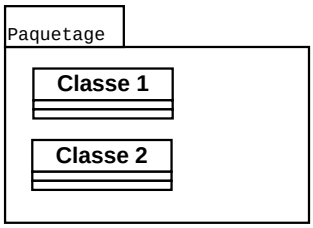

FIG. B.1 – Modélisation UML d'un paquetage.

Les paquetages divisent et organisent les modèles de la même manière que les répertoires organisent les systèmes de fichiers. Chaque paquetage correspond à un sous-ensemble du modèle et contient, selon le modèle, des classes, des objets, des relations, des composants ou des noeuds, ainsi que les diagrammes associés.

B.2 Classes et Visibilité

Une classe est le descripteur d'un ensemble d'objets qui ont une structure, un comportement et des relations similaires.

La figure B.2 montre la notation graphique d'une classe.

Les classes sont représentées par des rectangles compartimentés. Le premier compartiment contient le nom de la classe. Le second compartiment contient les attributs et le troisième contient

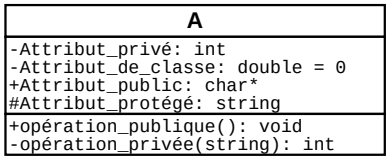

FIG. B.2 – Modélisation UML d'une classe.

les opérations.

UML définit trois niveau de visibilité pour les attributs et les opérations :

public qui rend l'élément visible à toutes les entités du programme,

protégé qui rend l'élément visible aux sous-classes de la classe,

privé qui rend l'élément visible à la classe seule.

Le niveau de visibilité est symbolisé par les caractères +, # et - qui correspondent respectivement aux niveaux public, protégé et privé.

Les attributs et opérations de classe, visibles globalement dans toute la portée lexicale de la classe, sont représentés par un nom souligné.

B.3 Associations et cardinalité

Les associations représentent des relations structurelles entre classes d'objets. La plupart des associations sont binaires et sont symbolisées par une ligne entre les classes associées.

Les associations peuvent être nommées. Le nom apparaît alors au milieu de la ligne qui la symbolise. L'extrémité d'une association est appelée rôle. Le rôle décrit comment une classe voit une autre classe au travers d'une association.

Chaque rôle d'une association porte une indication de multiplicité qui montre combien d'objets de la classe considérée peuvent être liés à un objet de l'autre classe. On trouve les cardinalités suivantes :

1 : un et un seul,

0..1 : zéro ou un,

M..N : de M à N,

***** : de zéro à plusieurs,

0..* : de zéro à plusieurs,

1..* : de un à plusieurs.

La figure B.3 montre la représentation graphique d'une association entre classes.

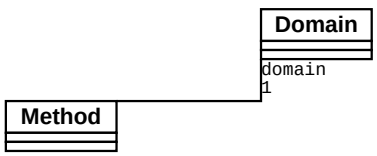

FIG. B.3 – Modélisation UML d'une association entre classes.

On trouve également des associations particulières dans lesquelles une des extrémités joue un rôle prédominant par rapport à l'autre : il s'agit des agrégations. Elle se représente par un petit losange du coté de l'agrégat.

Les critères suivant impliquent une agrégation :

– une classe fait partie d'une autre classe,
– les valeurs d'attributs d'une classe se propagent dans les valeurs d'attributs d'une autre classe,
– une action sur une classe implique une action sur une autre classe,
– les objets d'une classe sont subordonnés aux objets d'une autre classe.

La contenance physique est un cas particulier de l'agrégation appelé composition et se représente dans les diagrammes par un losange de couleur noire.

La composition implique une contrainte sur la valeur de la multiplicité du côté de l'agrégat : elle ne peut prendre que les valeurs 0 ou 1, la valeur 0 correspondant à un attribut non renseigné.

La figure B.4 illustre la représentation graphique d'une agrégation et d'une composition.

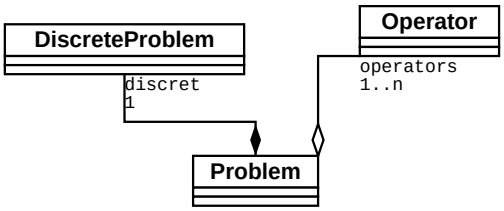

FIG. B.4 – Modélisation UML d'une agrégation et d'une composition.

B.4 Généralisation et héritage

La généralisation désigne la relation de classification entre un élément plus général et un élément plus spécifique. L'élément plus spécifique peut contenir des informations qui lui sont propres à condition de rester cohérent avec la description de l'élément plus général.

La relation de généralisation se représente au moyen d'une flèche qui pointe de la classe la plus spécialisée vers la classe la plus générale.

On dit aussi que la classe la plus spécialisée dérive ou hérite de la classe la plus générale. La relation de spécialisation signifie "est un" ou "est une sorte de".

La figure B.5 montre la représentation graphique d'une relation d'héritage.

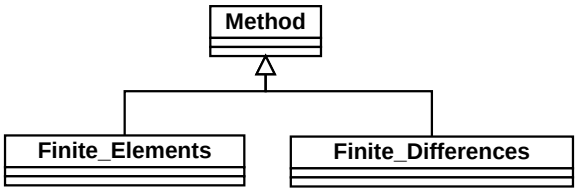

FIG. B.5 – Modélisation UML d'une relation de généralisation.

B.5 Conclusions

Nous avons présenté une description de la notation UML des diagrammes de classe qui correspondent à une vue de la structure statique du problème à modéliser. Cela ne peut en aucun cas être une vue globale de ce langage de modélisation, ni même une vue particulière sur le modèle statique.

Annexe C

CORBA : mécanismes de fonctionnement

C.1 Concepts et notions

La norme CORBA est née des réflexions d'un consortium international de l'industrie de l'inormatique crée en 1989 : l'OMG (Object Management Group). L'objectif de ce groupe est de faciliter l'émergence de standards pour les applications utilisant les technologies orientés objets dans un contexte d'environnements distribués hétérogènes.

L'OMG se fixe ainsi pour but le développement d'applications distribuées dont les composants collaborent avec :

- Efficacité,
- Fiabilité,
- Transparence,
- Scalability, c'est-à-dire une capacité d'évolution importante.

Les spécifications produites par l'OMG sont définies dans l'OMA (*Object Management Architecture*) qui correspond au modèle de référence et précise les concepts et l'architecture. Son noyau s'appelle *CORBA*.

Les principes de CORBA sont :

- Une séparation stricte Interface/Implémentation,
- La transparence de la localisation des objets,
- La transparence de l'accès aux objets,
- Le typage des Références d'Objets par les interfaces,
- L'héritage multiple d'interfaces.

Les spécifications de l'OMA s'appuient sur l'approche objet, qui a pour objectif de réduire les phases de développement et de maintenance des logiciels et permet de définir les concepts suivants :

- Interopérabilité : il faut assurer l'indépendance des mécanismes et des formats de données vis à vis des plates-formes, des systèmes d'exploitation, des couches réseau, du langage de programmation. CORBA assure l'interconnexion entre les objets distribués.
- Architecture client/serveur : un objet (objet serveur) offre des services à des applications clientes distantes (ou à d'autres objets, objets clients). L'objet serveur possède l'intégralité des caractéristiques de l'objet, le client accédant à ces services grâce aux méthodes de l'objet serveur.
- Bus logiciel : analogie avec les bus hardware où différents modules électroniques (carte mère, mémoire, périphériques...) interagissent via un bus et des interfaces (modularité, extensibilité).

L'OMA vise à clarifier les différents objets qui interviennent dans une application en fonction de leur rôle selon la figure C.1.

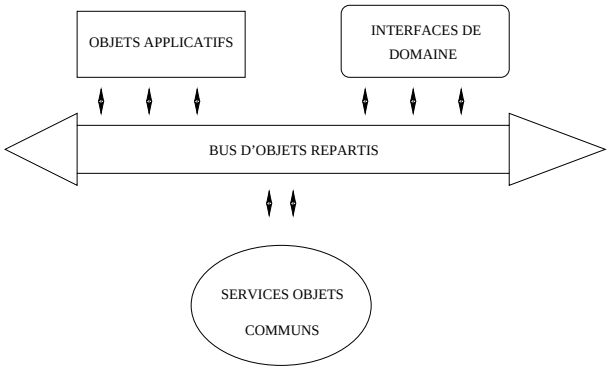

FIG. C.1 – Constitution de l'OMA.

Le bus d'objets répartis : il permet la communication entre clients et objets et assure ainsi le transport des requêtes entre tous les objets CORBA. Il offre un environnement d'exécution aux objets en masquant les hétérogénéités liées aux langages de programmation, aux systèmes d'exploitation, aux processeurs et aux réseaux.

Les services objets communs (CORBA Services) : ils fournissent sous forme d'objets CORBA les fonctions systèmes nécessaires à la plupart des applications réparties : annuaires (Naming Service), cycle de vie des objets, évènements (Event Service), sécurité, persistance ...

Les interfaces de domaines (Domain Interfaces) : elles définissent des objets métiers spécifiques à certains secteurs d'activités (santé, télécoms, finances ...).

Les objets applicatifs (Application Objects) : ils correspondent à des objets spécifiques à une application répartie et donc non standardisés.

Nous allons par la suite faire appel à un certain nombre de termes propres au vocabulaire CORBA que nous définissons donc maintenant.

IDL (Interface Definition Language) : Le langage IDL permet de définir les interfaces des objets CORBA. Il est indépendant des langages d'implémentation des servants de ces objets. La compilation d'une interface IDL génère un fichier souche (stub) pour la partie client et un fichier squelette (skeleton) pour la partie serveur.

Développer des applications distribuées flexibles sur des plates-formes hétérogènes nécessite une séparation stricte interface/implémentation. IDL aide à accomplir cette séparation.

C'est un langage de définition d'interface orienté objet. Il définit les types des objets en spécifiant leurs interfaces. Une interface consiste en un jeu d'opérations et de paramètres pour ces opérations.

IDL est le moyen par lequel une implémentation d'un objet indique à ses clients potentiels quelles opérations sont disponibles et comment elles doivent être invoquées.

IDL a été conçu pour assurer la correspondance avec des langages de programmation.

Les stubs (client) et les skeletons (serveur) automatisent les actions suivantes (en conjonction avec l'ORB) :
- codage/décodage des paramètres,
- génération des implémentations des classes d'interfaces,
- enregistrement et activation des objets,
- localisation et liens des objets.

Objet CORBA : entité virtuelle capable d'être localisée par un orb et recevant des requêtes des clients.

Objet cible (Target Object) : dans le contexte d'une requête CORBA, c'est l'objet cible de cette requête.

Client : entité qui invoque une requête sur un objet CORBA.

Serveur : application dans laquelle existent un ou plusieurs objets CORBA.

Requête (Request) : invocation d'une opération sur un objet CORBA par un client. La requête va du client à l'objet cible via le serveur et la réponse est aussi renvoyé de l'objet vers le client via le serveur.

Référence d'objet (Object Reference) : elle permet d'identifier, de localiser et de s'adresser à un objet CORBA. Les clients ne peuvent pas créer de référence d'objet. Une référence ne réfère qu'à un seul objet. Elles peuvent décrire des objets implémentés dans différents processus et sur différentes machines aussi bien que des objets implémentés sur le client lui-même.

La référence sur l'objet est le seul moyen pour le client d'atteindre un objet cible. Pour publier une référence, le serveur a plusieurs solutions possibles :
- Retourner une référence comme résultat d'une opération,
- Publier la référence dans un service commun (le Naming Service par exemple),
- Publier la référence en la convertissant en chaîne de caractères et en l'écrivant dans un fichier,
- Transmettre la référence par un autre mécanisme comme l'envoyer par e-mail ou la publier dans une page web.

Le moyen le plus commun est d'acquérir la référence sur l'objet en réponse à l'invocation d'une opération, celle-ci retournant une ou plusieurs références sur des objets. Les clients peuvent ainsi "naviguer" comme en suivant des liens hypertextes : l'obtention d'une référence permet par l'invocation d'une méthode sur cette référence d'en obtenir une autre et ainsi de suite.

Lorsqu'une référence est reçue par un client, l'exécutable du client instancie un objet proxy dans l'espace d'adressage du client. Lorsque le client invoque une opération sur le proxy, celui-ci envoie un message au servant distant. La classe Proxy est générée à partir de l'interface IDL et implémente le stub à travers lequel le client envoie ses appels.

Servant : entité de programmation implémentant un ou plusieurs objets CORBA. Ils incarnent les objets CORBA.

Object Request Broker (ORB) : noyau de communication. Il assure le transport des invocations entre les objets. Plusieurs protocoles sont standardisés (GIOP, IIOP ...). L'ORB fournit les mécanismes par lesquels les objets font des requêtes et reçoivent des réponses, et ce de manière transparente. Il fournit également l'interopérabilité entre des applications sur différentes machines dans des environnements distribués hétérogènes et il interconnecte sans coutures de multiples systèmes objets. D'une façon simplifiée, on peut définir l'ORB comme une entité qui fournit des mécanismes d'interrogations permettant de récupèrer des objets, des procédures qui constituent une application.

L'ORB est responsable de tous les mécanismes nécessaires pour :
- Trouver l'implémentation de l'objet pour la requête,
- Préparer cette implémentation à recevoir la requête,
- Communiquer les données constituant la requête.

Static Invocation Interface (SII) et Static Skeleton Interface (SSI) : Elles correspondent à l'invocation et au retour statique des requêtes. Dans ce cas, les interfaces IDL sont traduites en stubs et skeletons de manière langage dépendante et ceux-ci sont compilés dans les applications clientes et serveurs. Le stub est le côté client qui permet l'invocation d'une requête comme un appel local. Le skeleton est le côté serveur qui permet à une invocation reçue par le serveur de l'envoyer au servant approprié.

Dynamic Invocation Interface (DII) et Dynamic Skeleton Interface (DSI) : La construction et l'envoi des requêtes se fait à l'exécution plutôt qu'à la compilation. Cette méthode

nécessite l'accès à des services qui peuvent donner des informations sur les interfaces et les types. L'application peut obtenir ces informations par l'intermédiaire de l'Interface Repository, un service permettant l'accès aux définitions IDL au moment de l'exécution.

Pour invoquer une opération sur un objet, un client doit faire appel, et être lié statiquement au stub correspondant. Puisque le développeur détermine, à l'écriture du code, les stubs qu'un client contient, l'invocation statique ne peut accéder à de nouveaux objets qui ont été ajoutés au système plus tard. La DII fournit cette capacité. Cette interface permet à un client, à l'exécution, de :
- Découvrir de nouveaux objets,
- Découvrir leurs interfaces,
- Retrouver les définitions d'interfaces,
- Construire et distribuer des invocations,
- Recevoir les réponses,

et ceci de la part d'objets dont les stubs du client ne sont pas liés dans son module.

La DII est donc une interface de l'ORB qui comprend des routines autorisant le client et l'ORB, travaillant ensemble, à construire et invoquer des opérations sur tout objet disponible à l'exécution.

Object Adapter (OA) : Un adaptateur d'objet est l'interface principale pour une implémentation objet pour accéder aux services fournis par un ORB.

Il adapte l'objet CORBA au langage de la projection, gère et interprète les références d'objets et active les implémentations des servants lors des invocations.

Les adaptateurs d'objets servent de "glue" entre les servants et l'ORB. Du point de vue architectural de la conception orienté objet, un object adapter est un objet qui adapte l'interface d'un objet à une interface différente attendue par le requêteur.

Les adaptateurs d'objets :
- créent les références d'objets qui permettent aux clients de s'adresser à ces objets,
- s'assurent que chaque objet est incarné par un servant,
- résolvent les requêtes du coté serveur de l'ORB et les redirigent au servant approprié.

Il faut enregistrer les servants sur l'Object Adapter pour permettre le renvoi des requêtes.

Implementation Repository : C'est le référentiel des implémentations. Il contient l'information nécessaire à l'activation des objets.

Interface Repository (IR) : C'est le référentiel des interfaces. Il contient toutes les définitions des interfaces des objets du bus (définitions IDL). Ces informations sont accessibles par les mécanismes d'invocations statiques et dynamiques.

L'IR est donc le composant de l'ORB qui fournit un stockage persistant des définitions d'interfaces, il gère et permet l'accès à une collection de définitions d'objets spécifiés en IDL.

L'ORB peut utiliser les définitions d'objets contenues dans l'IR pour interpréter/manipuler les valeurs fournies lors d'une requête :
- Pour permettre la vérification du type des signatures des requêtes,
- Pour aider à fournir l'interopérabilité entre différentes implémentations de l'ORB.

ORB Interface : Il permet d'accéder à certaines fonctionnalités du bus : initialisation, paramétrage de l'environnement CORBA, outils de manipulation des références (instanciation, conversion...) et gestion de la mémoire.

GIOP, IIOP, ... : Le GIOP (General Inter-ORB Protocol) permet à deux ORB de communiquer. L'IIOP (Internet Inter-ORB Protocol) spécifie comment le GIOP est implémenté par dessus le protocole TCP/IP.

L'interopérabilité de l'ORB nécessite de standardiser les formats des références des objets par l'intermédiaire de l'Interoperable Object Reference (IOR).

Pour l'IIOP, l'IOR est constitué du nom de l'hôte, d'un port TCP/IP et d'une clé objet identifiant l'objet cible sur l'hôte et le port donné.

Compte tenu des notions générales que l'on vient de décrire, le développement d'une application CORBA de type client/serveur suit les étapes ci-dessous :

1. Déterminer les objets de l'application et définir leurs interfaces IDL.
2. Compiler les définitions IDL en stubs et skeletons.
3. Déclarer et implémenter les classes servants qui incarnent les objets CORBA,
4. Ecrire un programme principal pour le serveur :
   - initialisation de l'ORB et du POA (*Portable Object Adapter*),
   - création de servants,
   - déclaration de ces servants dans le POA pour faire en sorte que les servants incarnent les objets CORBA,
   - attendre les requêtes.
5. Compiler et linker l'implémentation du serveur avec les stubs et les skeletons et créer l'exécutable serveur.
6. Ecrire, compiler et linker le code client avec les stubs :
   - obtention des références des objets,
   - invocation d'opération sur les objets CORBA.

C.2 Fonctionnement du bus d'objets répartis et du processus d'invocation des requêtes

Le bus CORBA est le noyau de communication à travers lequel les objets vont dialoguer. Il a les caractéristiques suivantes :

- La liaison avec beaucoup de langages de programmation : actuellement, l'OMG a défini cette liaison pour les langages C, C++, Java, SmallTalk, Ada, COBOL.
- La transparence des invocations : les requêtes aux objets semblent toujours être locales, le bus CORBA se chargeant de les acheminer en utilisant le canal de communication le plus approprié.
- L'invocation statique et dynamique : ces deux mécanismes complémentaires permettent de soumettre les requêtes aux objets. En statique, les invocations sont contrôlées à la compilation. En dynamique, les invocations doivent être contrôlées à l'exécution.
- Un système auto-descriptif : les interfaces des objets sont connues du bus et sont aussi accessibles par les programmes par l'intermédiaire du référentiel des interfaces.
- L'activation automatique et transparente des objets : les objets sont en mémoire uniquement s'ils sont utilisés par des applications clientes.
- L'interopérabilité entre bus : un protocole générique de transport des requêtes (GIOP) a été défini permettant l'interconnexion de bus CORBA provenant de fournisseurs distincts, une de ses instanciations est l'IIOP fonctionnant au dessus de TCP/IP.

Les clients manipulent les objets en envoyant des messages c'est-à-dire en invoquant une opération sur l'objet. Pour envoyer un message à un objet, le client doit connaître une référence sur cet objet. Le processus d'invocation des requêtes est illustré par la figure C.2.

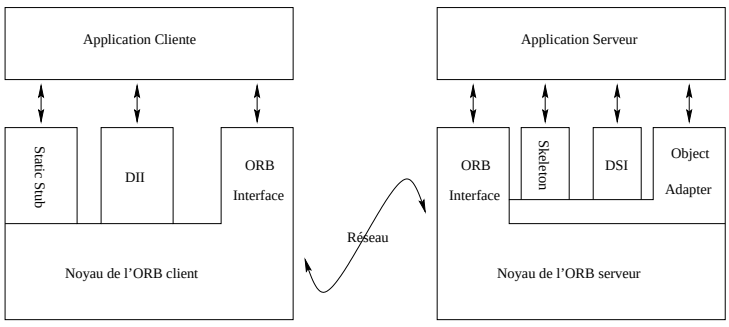

FIG. C.2 – Processus d'invocation des requêtes.

Les requêtes vont de l'application cliente via l'ORB vers l'application serveur :

1. Le client peut faire une requête en utilisant un "talon" (stub) statique ou en utilisant la *Dynamic Invocation Interface*.
2. Le noyau de l'ORB client localise l'objet cible. Il active le serveur si celui-ci n'est pas déjà en cours d'exécution puis il transmet les arguments et la requête au noyau de l'ORB serveur.
3. Le noyau de l'ORB serveur renvoie la requête à l'adaptateur d'objets (Object Adapter) qui active l'objet cible.
4. L'adaptateur d'objets renvoie la requête au servant implémentant l'objet cible. Comme le client, le serveur peut choisir entre un envoi statique ou dynamique : le servant utilise alors la *Dynamic Skelton Interface*.
5. L'ORB retourne tous les arguments de type out et inout lorsque l'appel de la méthode est terminé, ou retourne une exception si l'appel a échoué.

Pour le client, l'invocation d'une requête a les caractéristiques suivantes :
- localisation transparente,
- serveur transparent : le client n'a pas besoin de connaître le serveur implémentant l'objet,
- indépendance du langage entre client et serveur,
- indépendance de l'implémentation : le client ne sait pas comment l'objet est implémenté,
- indépendance de l'architecture,
- indépendance de l'OS,
- indépendance du protocole,
- indépendance du transport.

C.3 Utilisation des services CORBA

Un service est caractérisé par les interfaces qu'il fournit et par les objets qui fournissent ces interfaces. Un service peut impliquer un seul objet, plusieurs objets qui fournissent le même type d'interface ou plusieurs objets qui fournissent des types d'interfaces distincts qui héritent d'un type d'interface de service.

Chaque ObjectService fournit son service à un jeu d'utilisateurs qui sont typiquement des applications, mais qui peuvent être d'autres ObjectServices.

Nous ne décrirons ici que les services utilisés dans notre projet.

C.3.1 Le service de Nommage (Naming Service)

Le service de nommage fournit le principal mécanisme à travers lequel la plupart des objets d'un système basé-ORB situent les objets qu'ils veulent utiliser.

Il équivaut aux "pages blanches" : les objets sont désignés par des noms symboliques et l'obtention des références d'objets ne nécessite plus de passer par un support de chaîne de caractères (fichier, e-mail ...).

Il permet, à partir d'un nom, d'obtenir une référence sur l'objet correspondant. C'est l'équivalent du DNS qui traduit des noms de domaine internet en adresse IP.

Le service de nommage permet aux applications :
- L'utilisation des noms parlant pour les objets au lieu de référence sous forme de chaîne de caractères.
- Le changement de la valeur d'une référence sur un objet conservant le même nom permet de ne pas modifier le code de l'application.
- Il n'est plus nécessaire de stocker les références sur fichier pour la résolution initiale (connexion à l'ORB).

Une association nom-référence est appelée *n*ame binding. Un contexte de nom (naming context) est un objet qui stocke les *n*ame bindings : chaque objet contexte implémente une table qui associe des noms à des références. Un nom dans une table peut désigner aussi bien une référence d'objet sur un objet propre à une application qu'un autre contexte du service de nommage. Le contexte est donc un jeu de liens noms-objets dans lequel chaque nom est unique.

Une hiérarchie de contextes et d'associations nom-référence est appelé un graphe de nommage (naming graph). Le graphe de nommage a les caractéristiques suivantes :
- Pour un contexte particulier, les associations nom-références sont uniques.
- Un objet ou un contexte peut avoir plusieurs noms, à la manière des notions des liens de fichier ou de répertoire sous UNIX.
- Il existe des contextes sans nom : ils sont dits orphelins.
- Un graphe de nommage a des contextes particuliers appelés *i*nitial naming contexts, qui correspondent en général aux contextes orphelins et qui équivalent aux répertoires racine d'UNIX.

Le service de nommage permet :
- de lier un nom à un objet, ceci dans un contexte de nommage,
- de résoudre un nom, c'est-à-dire déterminer l'objet associé au nom dans un contexte donné.

C.3.2 Le service des Evènements (Event Service)

Dans CORBA, l'invocation standard d'une méthode consiste en une exécution synchrone d'une opération fournie par un objet. Or, pour la plupart des applications, un modèle de communication plus découplé entre objets est nécessaire. L'OMG a donc défini un jeu d'interfaces eventservice qui permettent la communication asynchrone entre objets. Le modèle de l'OMG est basé sur le paradigme publish/subscribe.

L'Event Service permet donc aux objets de produire des évènements asynchrones à destination d'objets consommateurs à travers des canaux d'évènements.

Dans le modèle du service d'évènements, les fournisseurs produisent les évènements et les consommateurs les reçoivent. Les deux sont reliés par le canal d'évènement (Event Channel). Celui-ci convoit les évènements des fournisseurs aux consommateurs sans que ceux-ci ne se connaissent.

Il est responsable de l'enregistrement des fournisseurs et consommateurs, de l'envoi des évènements à tous les consommateurs enregistrés et du traitement des erreurs associées aux consommateurs ne répondant pas.

Les canaux d'évènements permettent à plusieurs fournisseurs et consommateurs de se connecter et peut gèrer différents modes de fonctionnement (modèle push, pull et hybrides) :

Le modèle push : Les fournisseurs poussent les évènement dans le canal qui à son tour les poussent vers tous les consommateurs enregistrés. Les fournisseurs sont donc les initiateurs actifs des évènements et les consommateurs attendent passivement de les recevoir. Le canal d'évènement joue le rôle du notifieur.

Le modèle pull : Les consommateurs tirent les évènements du canal, qui à son tour les tirent des fournisseurs. Les consommateurs sont les initiateurs actifs des évènements, et les fournisseurs attendent passivement que les évènements leur soient demandés. Le canal joue le rôle du procureur.

Le modèle hybride push/pull : Les fournisseurs poussent les évènements dans le canal où ils sont tirés par les consommateurs. Les deux sont donc actifs. Le canal joue le rôle de pile car il stocke les données des évènements en attendant qu'elles soient tirées par les consommateurs.

Le modèle hybride pull/push : Le canal tire les évènements des fournisseurs et les poussent vers les consommateurs. Les deux sont passifs et le canal a le rôle d'agent intelligent.